Towards Parallel Execution of Sequential Scientific Applications

Kristensen, Mads Ruben Burgdorff

Publication date:
2012

Document version
Early version, also known as pre-print

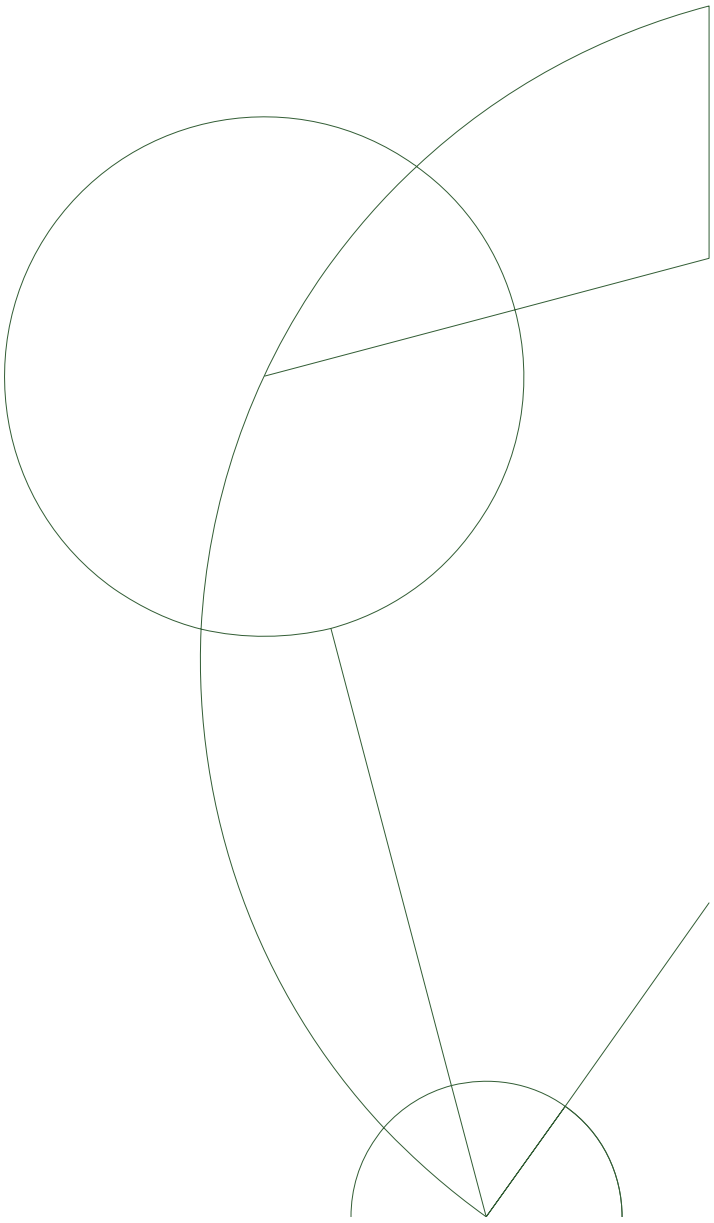
Citation for published version (APA):
Kristensen, M. R. B. (2012). *Towards Parallel Execution of Sequential Scientific Applications*. (Københavns Universitet, Niels Bohr Institute ed.).

FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN
Niels Bohr Institute
Denmark

PhD thesis

Mads Ruben Burgdorff Kristensen

Towards Parallel Execution of Sequential Scientific Applications



Academic advisor: Brian Vinter

Submitted: June 1, 2012

Acknowledgments

Firstly, I would like to thank all my coworkers at the eScience Center and my fellow PhD-students who made my PhD study a very nice experience. Especially, I would like to thank my supervisor Professor Brian Vinter for whom this thesis would not have been possible. Thanks for this educational and rewarding experience and the opportunity to attend several international conferences and Ph.D. summer schools.

I would like to thank Hans H. Happe for his supervising when I started the PhD study. It was a great help to have a mentor, in addition to Brian, in the initial phase of my study.

During the first year of my PhD, I had the privilege to work with the GPAW team at Technical University of Denmark – special thanks to Jens. J. Mortensen for great collaboration and hospitality.

In the spring of 2011, I moved to Berkeley to visit Professor Katherine Yelick and her research group at Lawrence Berkeley National Laboratory for five months. I would like to thank Katherine Yelick for the warm welcome and the opportunity to work with her group. Additionally, I would like to thank Yili Zheng for his interest in my work and for co-authoring one of the published papers.

I would like to thank Simon A. F. Lund and Troels Blum for great collaboration in the cphVB project. I hope that we can continue our work together to make the cphVB project even better.

Finally, I would like to thank my family and girlfriend, Mette, who all have been very understanding and supportive.

This research is supported by the Danish Strategic Research Council, grant #09-063770.

Abstract

Rapid prototyping of numerically expressed problems is essential for a broad range of research areas. Finding the solution for computational scientific and engineering problems often requires experimenting with various algorithms and different parameters using the feedback from several iterations. Therefore, being able to quickly prototype the solution is critical for a timely and successful scientific discovery.

High-productivity languages such as Matlab and Python are popular languages in the scientific community because of the need for rapid prototyping. However, they are generally accepted as being much slower than compiled languages, such as C or FORTRAN. More importantly, since they are inherently sequential which makes them unsuitable for prototyping on large data sets.

In this thesis, I have explored the possibility of seamlessly executing sequential scientific applications in parallel. Essential for my work is the vector-oriented programming model as a high-productivity programming approach to develop applications that targets a broad range of parallel hardware architectures. The idea is to introduce implicit data parallelism in order to provide a high-productivity and high-performance framework.

I introduce two new projects, DistNumPy and cphVB, that strives to provide a high-performance back-end for Numerical Python (NumPy) without reducing the high-productivity of Python/NumPy. The DistNumPy project targets distributed memory architectures and utilize automatic communication-latency hiding.

The cphVB project generalizes the design of DistNumPy to support a broad range of languages and hardware architectures. The implementation of cphVB consists of a language frontend that translates language specific array operations into cphVB vector operations. The frontend will send these vector operations to a Vector Engine that performs the actual execution of the operations. The design of cphVB support a broad range of Vector Engines that are optimized to specific hardware architectures, such as multi-core CPUs, GPGPUs and clusters of said architectures. For all languages that have a cphVB frontend, cphVB provides a high-productivity, high-performance framework that seamlessly parallelizes legacy applications without changing a single line of code.

I present several performance studies that demonstrate good scalable performance on a variety of architectures: from a small Ethernet Linux cluster with 32 CPU-cores to the Cray XE-6 supercomputer *Hopper* with 1536 CPU-cores.

Resumé

Computerberegninger er blevet en essentiel del af naturvidenskaben og computerprogrammering er nu også et vigtigt værktøj for forsker uden en datalogisk baggrund. Forskere vil ofte programmere prototype programmer som en metode til at udvikle og evaluere forskellige algoritmer. Netop derfor er programmeringssprog der fokusere på høj produktivitet, som fx Matlab og Python, yders populære i naturvidenskaben. De er dog generelt accepteret som værende langsommere end kompileret programmeringssprog som fx C eller FORTRAN. Ydermere er de som udgangspunkt sekventielle og supportere ikke parallel programmering hvilket besværliggør beregning med store data sæts der kræver flere maskiner.

I denne afhandling undersøges muligheden for at eksekvere sekventielt program parallelt helt automatisk. Den vektororientere programmeringsmodel bliver ofte brugt i naturvidenskaben da programkoden i høj grad ligner de tilsvarende matematiske udtryk. Det udnyttes ved at introducere implicit data parallelitet og optimere eksekvering til en lang række forskellige computerarkitekturer.

I denne afhandling præsenterer jeg to projekter, DistNumPy og cphVB, hvis hovedformål er at levere høj programmeringsproduktivitet og høj beregningsydelse. DistNumPy er en *backend* til the numeriske Python bibliotek NumPy der introducere implicit parallelitet uden at reducere NumPy's programmeringsproduktivitet. DistNumPy kan køre på distribueret hukommelse maskiner og udføre automatisk *communication-latency hiding*. I cphVB projektet generaliseres DistNumPy designet så det også understøtter andre programmeringssprog og computer arkitekturer.

Jeg præsenterer ydelsesevalueringer som demonstrerer god skalerbar ydelse på flere forskellige maskiner: fra en lille Ethernet Linux klynge med 32 CPU-kerner til Cray's Supercomputer *Hopper* hvor 1536 CPU-kerner udnyttes.

Contents

Chapter 1

Introduction

As computers have evolved, computer simulations have become an important part of natural sciences. Computer simulations make it possible to conduct experiments that would otherwise be impractical, e.g. simulating an earthquake or a nuclear detonation. A computer simulation may become such an extensive computational task that a number of processors is needed for the simulation to finish in reasonable time. Typically, it is also the case that the memory requirement alone makes it mandatory to distribute the computation between multiple processors. The development of such simulations requires high expertise in the relevant scientific area thus the development and implementation is often done by non-computer experts. The Holy Grail for many scientific frameworks is therefore to ease the programming, increase the productivity and support efficient parallelization.

The development of numerical simulations often consists of two implementations: a prototype and a final version. The algorithm is developed and implemented in a prototype by which the correctness of the algorithm can be verified. Typical many iterations of development are required to obtain a correct prototype, thus for this purpose a high productivity language is used, such as MATLAB[52]. However, when the correct algorithm is finished the performance of the implementation becomes essential for achieving results.

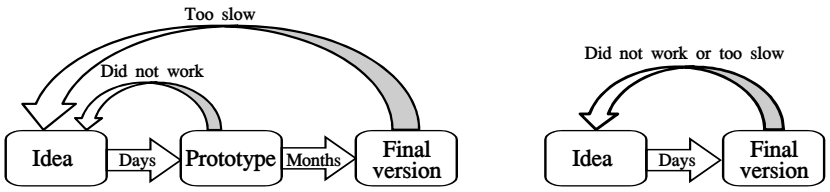


Figure 1.1: Development workflow. (a) is a typical workflow that involves two languages: one for the prototype and one for the final version. In (b) only one language is used in the workflow.

This performance requirement presents a problem for the researcher since highly optimized code requires a fairly low-level programming language such as C/C++ or Fortran. The final version will therefore typically be a reimplementation of the prototype, which involves both changing the programming language and parallelizing the implementation (Figure 1.1a).

The overall target of this work is to provide a high productivity tool that meets both the need for a high productivity tool that allows researcher to move from idea to prototype in a short time, and the need for a high performance solution that will eliminate the need for a costly and risky reimplementation (Figure 1.1b). It should be possible to develop and implement an algorithm using a simple notebook and then effortlessly execute the implementation on a cluster of computers while utilizing all available CPUs.

My approach to achieve this goal has been twofold. First obtain experience in optimizing a concrete scientific simulation for a massive parallel architecture and then develop are framework that easy the process of implementing such scientific simulations.

1.1 Scientific Expressions

The primary work when implementing scientific applications is the discipline of programming mathematical expressions. Thus, it is essential that the programming language support high-level numerical structures such as vector and matrices. Furthermore, high-level operations performed directly on vector and matrices come as a natural requirement.

As an example, consider the expression of a Jacobi Iteration. When A is a symmetric matrix with n^2 elements we can express one Jacobi Iteration as follows [1].

$$A_{x,y} = \frac{A_{x,y} + A_{x-1,y} + A_{x+1,y} + A_{x,y-1} + A_{x,y+1}}{5}, \{x \in \mathbb{N}_0,\ x < n,\ y \in \mathbb{N}_0,\ y < n\}$$

In order to be productive, the user needs a programming language where such a mathematical expression is easy to implement without the need for loop constructions or complex index or pointer arithmetic.

1.2 Contributions

The central contribution of this thesis is the introduction of *Distributed Numerical Python* (DistNumPy) and its generalization *Copenhagen Vector Bytecode* (cphVB) in addition to the optimization of a concrete scientific application, GPAW, for a massive parallel architecture.

GPAW In order to gain first-hand experience with the optimization of scientific applications, my initial work was to optimize GPAW – a simulation software that simulates

[1] Note that the expression assume that there exist meaningful boundary conditions to handle negative indices

many-body systems at the sub-atomic level – for the Blue Gene/P supercomputer. A team at the Technical University of Denmark drives the development of GPAW.

The focus of the work is to optimize a distributed stencil operation, which make up a substantial part of the overall execution time, using hybrid programming. Through a performance study, I evaluate four parallel programming models – two models that use hybrid programming and two that uses a flat programming model. Additionally, I evaluate the performance of my implementation of communication latency-hiding and communication batching.

The work has resulted in two published papers: *GPAW Optimized for Blue Gene/P using Hybrid Programming* and *Hybrid Parallel Programming for Blue Gene/P* where the latter paper is an extended version of the first paper.

DistNumPy In order to provide a high-productivity high-performance framework for scientific applications we introduce DistNumPy – a library for doing numerical computations in Python that targets scalable distributed memory architectures. DistNumPy is a new version of NumPy that enables the user to write sequential Python/NumPy applications that seamlessly utilize distributed memory architectures. DistNumPy implements data parallelism seamlessly by exploiting the vector-oriented programming model in NumPy. It uses a new backend for NumPy arrays that distribute data amongst the nodes in a distributed memory multi-processor. All operations on this new array will seek to utilize all available processors. The array itself is distributed between multiple processors in order to support larger arrays than a single node can hold in memory.

The work has resulted in four published papers: *Numerical Python for Scalable Architectures*, *Managing Overlapping Data Structures for Data-Parallel Applications on Distributed Memory Architectures*, *Managing Communication Latency-Hiding at Runtime for Parallel Programming Languages and Libraries*, and *PGAS for Distributed Numerical Python Targeting Multi-core Clusters*.

cphVB Contrary to DistNumPy, cphVB targets a broad range of programming languages and hardware architectures. In order to support multiple targets we generalize the design of DistNumPy. cphVB still uses the vector-oriented programming model to close the gap between high-level languages and hardware optimized low-level implementations. By translating high-level vector operations into an intermediate vector byte code, cphVB enables specialized vector engines to efficiently execute the vector operations.

The development of cphVB is in close collaboration with Troels Blum, who wrote his master thesis on the topic [17], and Simon A. F. Lund. No cphVB papers has been published yet but I have submitted one paper *cphVB: A Scalable Virtual Machine for Vectorized Applications*.

1.3 Publications

Mads Ruben Burgdorff Kristensen, Hans Henrik Happe, and Brian Vinter. GPAW Optimized for Blue Gene/P using Hybrid Programming
In Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing (IPDPS '09). IEEE Computer Society, Washington, DC, USA, 1-6.

M. Kristensen, H. Happe, and B. Vinter, Hybrid Parallel Programming for Blue Gene/P
Scalable Computing: Practice and Experience, vol. 12, no. 2, 2011. ISSN 1895-1767.

Mads Ruben Burgdorff Kristensen and Brian Vinter. Numerical Python for Scalable Architectures
In Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model (PGAS '10). ACM, New York, NY, USA

Mads Ruben Burgdorff Kristensen and Brian Vinter. Managing Overlapping Data Structures for Data-Parallel Applications on Distributed Memory Architectures
GSTF International Journal on Computing (JoC), vol. 1, no. 2, pp. 145-151, 2012. ISSN: 2010-2283.

Mads Ruben Burgdorff Kristensen and Brian Vinter. Managing Communication Latency-Hiding at Runtime for Parallel Programming Languages and Libraries
In Proceedings of the 2012 IEEE International Conference on High Performance Computing and Communications (HPCC'12). IEEE.

Mads Ruben Burgdorff Kristensen, Yili Zheng, and Brian Vinter. PGAS for Distributed Numerical Python Targeting Multi-core Clusters
In Proceedings of the 2012 IEEE International Symposium on Parallel and Distributed Processing (IPDPS'12). IEEE.

M. Kristensen, S. Lund, T. Blum, and B. Vinter. cphVB: A Scalable Virtual Machine for Vectorized Applications
Submitted to the International Conference on Parallel Processing (ICPP'12). Pittsburgh, PA, 2012.

1.4 Thesis Outline

This thesis is organized as follows: Chapter 2 and 3 provides background, Chapter 4 describe the GPAW project, Chapter 5 discuss programming productivity, Chapter 6 provides NumPy details, Chapter 7 and 8 presents the two key contributions DistNumPy

and cphVB, Chapter 9 contains the future work, and finally the conclusion is presented in Chapter 10. The papers that constitute the primary contribution of this thesis are included in Appendix A.

Chapter 2

Parallel programming

Parallel programming is the discipline of programming applications that makes use of multiple compute resources. There exist many levels of parallelism – from low-level parallelism where a single processor execute instructions in parallel to parallelism where multiple threads or processes executes the same or different instructions on the same or different data.

The process of parallelizing computer simulations in natural sciences often demands parallelism on many levels because of the huge amount of speedup required by scientists. In this work, we will focus on parallelism that uses multiple CPU-cores in a shared and/or distributed memory setup. We will only indirectly utilize lower-level parallelism within a single CPU-core, such as Streaming SIMD Extensions (SSE), through numerical libraries, e.g. LAPACK[71] and BLAS[7].

In this chapter, we will describe four essential parallel programming approaches: shared memory programming, distributed memory programming, hybrid programming and vector oriented programming.

2.1 Shared Memory Programming

The primary focus regarding shared memory programming is architectures that make use of symmetric multiprocessing (SMP) with cache coherency[48, 33]. It is a very common architecture and has become more popular with the trend of increasing the number of CPU-cores rather than the CPU-frequency. SMP is used in a long range of different architectures from the widespread x86 to the specialized Blue Gene architecture.

The main concept in the shared memory programming paradigm is the sharing of the main memory between different threads (or processes). It makes it possible for threads to independently work on common data structures and communicate through shared variables. Most operation systems support threads and provide constructs for controlling the flow of threads, such as mutex.

Most programming languages also support threading in some capacity. C and C++ do not provide direct support for threading on their own, but provide access to the native threading API provided by the operating system. Higher level programming

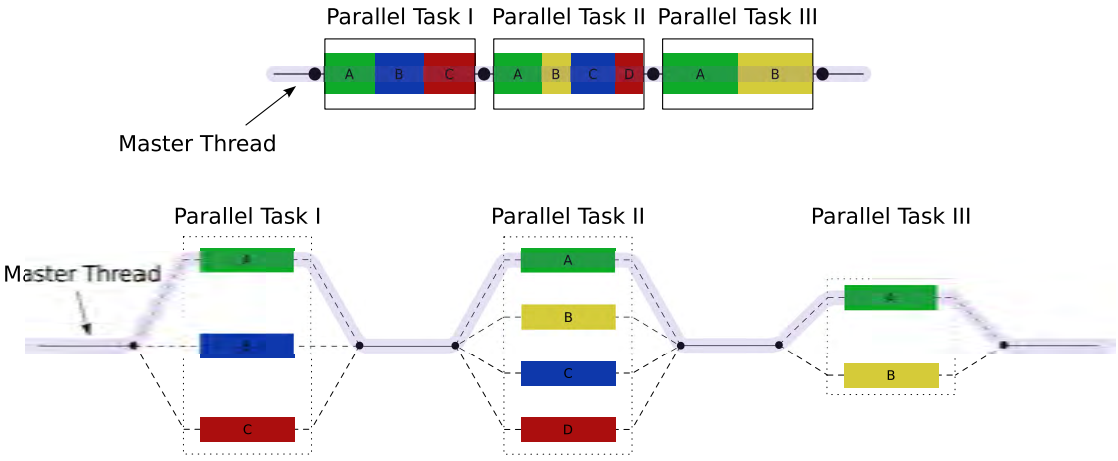


Figure 2.1: The Fork/Join parallel paradigm. The master thread forks off a number of threads which execute blocks of code in parallel.

languages, such as Java and Python, expose threading to the developer while abstracting the platform specific differences in the thread implementation.

A number of other programming languages and language extensions also try to abstract the concept of threading, and in some cases even parallelism, from the developer altogether.

2.1.1 Open Multi-Processing

Open Multi-Processing (OpenMP)[34, 84] is a multi-platform shared-memory parallel programming extension for C/C++ and Fortran. OpenMP makes the parallelization of an application easier by handling the tedious work involved in multi-threading programming, such as creating, joining and destroying threads. Additionally, OpenMP provide parallelization strategies for implementing common code structures, such as automatically parallelizing for-loops that have no dependent iterations. A broad range of compilers and platforms supports OpenMP.

OpenMP uses the Fork/Join parallel paradigm (Figure 2.1). An OpenMP program begins execution as a single process, called the master thread of execution. The fundamental directive for expressing parallelism is the `parallel directive`. It defines a parallel section of the program that is executed by multiple threads. When the master threads enters a parallel section, it forks a team of threads (one of them being the master thread), and work is continued in parallel among these threads. Upon exiting the parallel construct, the threads in the team synchronize (join the master), and only the master continues execution. The statements in the parallel section, including functions called from within the enclosed statements, are executed in parallel by each thread in the team. The user is required to keep this in mind when using OpenMP because all variables used in a parallel section must be classified as either shared or private. It should therefore be noted that OpenMP does not hide the parallelization from the user,

```
1  //Parameters
2  int I;      //Number of iterations
3  double *A; //Input & Output Matrix
4  double *T; //Temporary array
5  int SIZE;  //Symmetric Matrix Size
6
7  //Computation
8  int gsize = SIZE+2; //Size + borders.
9  for(n=0; n<I; n++)
10 {
11   memcpy(T, A, gsize*gsize*sizeof(double));
12   #pragma omp parallel for shared(A,T)
13   for(i=0; i<SIZE; ++i)
14   {
15     int a = i * gsize;
16     double *up     = &A[a+1];
17     double *left   = &A[a+gsize];
18     double *right  = &A[a+gsize+2];
19     double *down   = &A[a+1+gsize*2];
20     double *center = &T[a+gsize+1];
21     for(j=0; j<SIZE; ++j)
22       *center++ = (*center + *up++ + *left++ + *right++ + *down++) \
23                   / 5.0;
24   }
25   memcpy(A, T, gsize*gsize*sizeof(double));
26 }
```

Figure 2.2: Parallel version of Jacobi Iterations using OpenMP.

but provides convenient language constructs to handle parallelisms. Using OpenMP still requires knowledge in parallel programming and experience in the Fork/Join parallel paradigm.

Figure 2.2 shows an application written in C that implements the Jacobi Iteration from Section 1.1 and uses OpenMP for parallelization.

Performance OpenMP is a standard and it is therefore difficult to do any general reasoning about the performance aspects of OpenMP. However, work done in [43] demonstrates the scalability of seven applications executed on the Sun Fire 15K multiprocessor system. The applications are parallelized using OpenMP and the results show that five of the seven applications scale very well. Furthermore, it is shown that the overhead associated with OpenMP is kept at an acceptable level.

On the other hand very poor scalability is demonstrated in [59] where computation loops in Fire Dynamics Simulator is evaluated. The loops only scale to around two CPU-cores (Figure 2.3). The problem is the memory bound computation done inside the loops. OpenMP does not address this problem and it is therefore up to the developer and the processor architecture to handle.

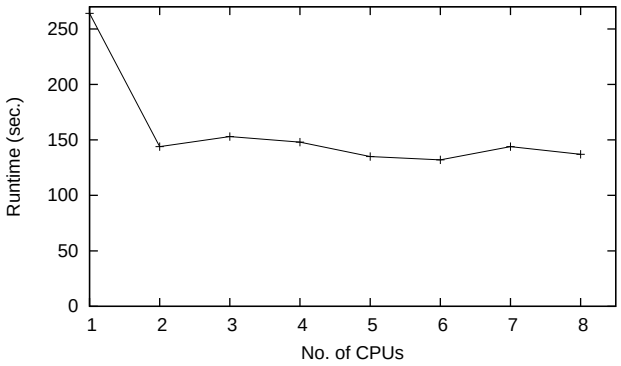


Figure 2.3: Runtime of computation loops parallelized using OpenMP on an eight core machine. The computation loops are part of a Computational Fluid Dynamics software called Fire Dynamics Simulator (From [59]).

2.1.2 Numerical Libraries

A great diversity of libraries supports some level of shared memory parallelism; some focusing on very specific problems and others on very general problems. The focus in this work is the two numerical libraries BLAS and LAPACK, which both support parallel execution for some implementations. They are both the *de facto* standard in the field of Computational Linear Algebra.

BLAS

Basic Linear Algebra Subprograms[71] is a very popular API standard for doing basic linear algebra operations such as vector and matrix multiplication. BLAS is heavily used in high performance computing and highly optimized implementations of the BLAS interface have been developed by hardware vendors such as Intel and IBM, as well as by other authors, e.g. Goto BLAS[**?**] and ATLAS[**?**]. Many of those implementations supports shared memory parallel execution. [36] demonstrates descent speedup of LU Factorization when parallelized by the use of parallel BLAS operations, but conclude that there is an unexploited potential of parallelism at the level above BLAS.

LAPACK

Linear Algebra Package[7] is a popular library for numerical linear algebra that provides routines for solving systems of linear equations, eigenvalue problems, and singular value decomposition. It also includes routines to implement the associated matrix factorizations such as LU, QR and Cholesky decomposition. LAPACK makes use of BLAS in order to effectively exploit the underlying hardware and possibly multiple CPU-cores.

The parallelization of a central operation in LAPACK, LU factorization, is demonstrated in [22] to yield better performance than a BLAS based parallel implementation.

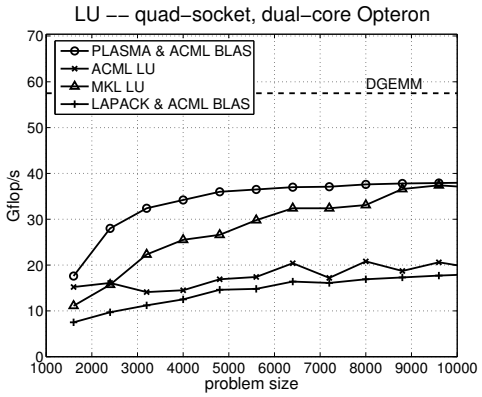


Figure 2.4: Performance of LU factorization executed on an AMD Opteron machine with a total of sixteen cores. ACML and MKL are the numerical libraries from AMD and Intel, respectively. PLASMA is the implementation developed in [22]. (From [22]).

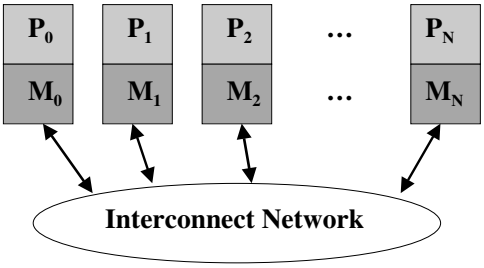


Figure 2.5: Distributed Memory.

The result is compared with a parallel implementation that only parallelizes the BLAS operations, and a numerical library from Intel and one from AMD (Figure 2.4).

2.2 Distributed Memory Programming

Distributed memory programming is used when implementing software for multiple processor systems in which each processor has its own private memory (Figure 2.5). Computational tasks can only operate on local data, and if remote data is required, the computational task must communicate with one or more remote processors. In this section, we will focus on some of the libraries that help developers to implement efficient distributed memory programming.

The very basic communication mechanism is the use of network sockets in which the data communication and manipulation is performed explicitly by the user. Communication through sockets is limited to either reading or writing a stream of bytes. All meta-data associated with the byte stream must be communicated explicitly, which means that basic tasks, such as determining whenever a whole message is received, needs to be handled by the user. Only the transport protocol is abstracted from the user.

Two communication paradigms that address inter-process communication are Message Passing and Remote Memory Access. In this context Remote Memory Access refers to the explicit use of operations that access remote memory, which is also known as one-sided communication. The fundamental difference between the two paradigms are whenever one or multiple processes are involved in the communication. In Message Passing at least two processes will initiate the communication with matching arguments. In Remote Memory Access only one part will initiate the communication with arguments for both the sending and receiving process.

2.2.1 Message Passing

Message Passing is often used with the SPMD (Single Process, Multiple Data) programming model because of the symmetry in the communication. That is, a process that send a message will normally also receive a similar message.

In my study I have investigated two popular message passing standards: PVM[94] and MPI[50]. The focus of PVM is networks of heterogeneous computers; whereas the focus of MPI is networks of homogeneous computers. Additionally, MPI applications uses the SPMD programming model almost exclusively; whereas PVM applications often use a more asynchronies communication model where processes have different roles, such as consumer or producer.

PVM

Parallel Virtual Machine[94] is a software system that enables a collection of heterogeneous computers to be used as one Distributed Memory Machine. PVM is built around the concept of a *virtual machine* which is a dynamic collection of (potentially heterogeneous) computational resources managed as a single parallel computer. One aspect of the virtual machine is how parallel tasks exchange data. This is accomplished using simple message passing constructs. The virtual machine transparently handles message routing and data conversion for incompatible architectures. However, when a message is send the user has to *pack* the message in a buffer, and likewise when receiving a message the user has to *unpack* it.

PVM supports a dynamic resource management. Computing resources can be added or deleted at will, either from a system console or even from within the user's application. Allowing applications to interact with and manipulate their computing environment provides a powerful paradigm for load balancing, task migration, and fault tolerance. The virtual machine provides a framework for determining which tasks are running and supports naming services so that independently spawned tasks can find each other and cooperate.

PVM supports a basic fault notification scheme. Under the control of the user, tasks can register with PVM to be notified when the status of the virtual machine changes or when task fails. A task can post a notify for any of the tasks from which it expects to receive a message. In this scenario, if a task dies, the receiving task will get a

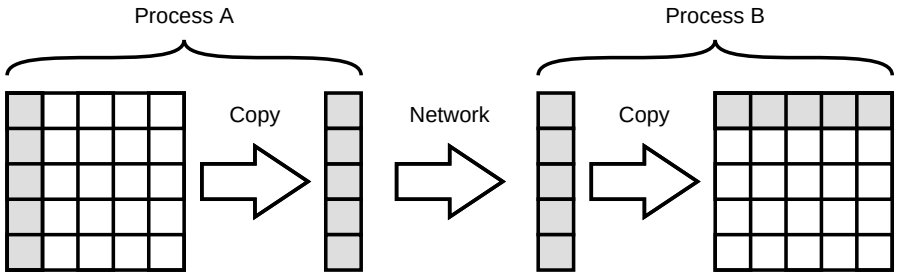


Figure 2.6: Illustration of the required message copies in MPI without user-defined data types. Process `A` send a matrix-column to process `B`, which interprets the message as a matrix-row. Note that if the matrix layout in this example happen to be column-major then `A` would not have to use a buffer and similar if the matrix layout is row-major the `B` would not have to use a buffer.

notify message in place of any expected message. The notify message allows the task an opportunity to respond to the fault without hanging or failing.

Similarly, if a specific host such as an I/O server is critical to the application, then the application tasks can post notifications for that host. The tasks will then be informed if that server exits the virtual machine, and they can allocate a new I/O server.

MPI

In contrast to PVM, Message Parsing Interface (MPI)[50] emphasize a more tightly bound communication paradigm in which a cluster of homogeneous nodes are preferred, though the possibility of using different architectures still exists..

To minimize the number of needed memory copies in a message passing the MPI standard supports user-defined data types, which makes it possible to send and receive non-contiguous data blocks. A typical scenario is one process who wants to send a column of a matrix to another process, and the other process wants to receive the column as a row in its own matrix (Figure 2.6). Without user-defined data types both processes must copy the matrix elements to and from a buffer. User-defined data types make it possible to avoid the use of buffers, but it should be noted that a MPI implementation is allowed to do copying anyway.

MPI does not provide mechanisms for dealing with failures in the communication system. If the MPI implementation is built on an unreliable underlying mechanism, then it is the job of the implementer of the MPI subsystem to insulate the user from this unreliability, or to reflect unrecoverable errors as failures. Similarly, MPI itself provides no mechanisms for handling processor failures.

Figure 2.7 shows an application written in C that implements the Jacobi Iteration from Section 1.1 and uses MPI for parallelization.

```
1  //Parameters
2  I; //Number of iterations
3  A; //Input & Output Matrix (local)
4  T; //Temporary array (local)
5  SIZE; //Symmetric Matrix Size (local)
6
7  //Computation
8  int gsize = SIZE+2; //Size + borders.
9  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
10 MPI_Comm_size(MPI_COMM_WORLD, &worldsize);
11 MPI_Comm comm;
12 int periods[] = {0};
13 MPI_Cart_create(MPI_COMM_WORLD, 1, &worldsize,
14                 periods, 1, &comm);
15 int l_size = SIZE / worldsize;
16 if(myrank == worldsize-1)
17    l_size += SIZE % worldsize;
18 int l_gsize = l_size + 2;//Size + borders.
19 for(n=0; n<I; n++)
20 {
21   int p_src, p_dest;
22   //Send/receive - neighbor above
23   MPI_Cart_shift(comm,0,1,&p_src,&p_dest);
24   MPI_Sendrecv(A+gsize,gsize,MPI_DOUBLE,
25                p_dest,1,A,gsize, MPI_DOUBLE,
26                p_src,1,comm,MPI_STATUS_IGNORE);
27   //Send/receive - neighbor below
28   MPI_Cart_shift(comm,0,-1,&p_src,&p_dest);
29   MPI_Sendrecv(A+(l_gsize-2)*gsize,
30                gsize,MPI_DOUBLE,
31                p_dest,1,A+(l_gsize-1)*gsize,
32                gsize,MPI_DOUBLE,
33                p_src,1,comm,MPI_STATUS_IGNORE);
34   memcpy(T, A, l_gsize*gsize*sizeof(double));
35   double *a = A;
36   double *t = T;
37   for(i=0; i<SIZE; ++i)
38   {
39     int a = i * gsize;
40     double *up     = &A[a+1];
41     double *left   = &A[a+gsize];
42     double *right  = &A[a+gsize+2];
43     double *down   = &A[a+1+gsize*2];
44     double *center = &T[a+gsize+1];
45     for(j=0; j<SIZE; ++j)
46       *center++ = (*center + *up++ + *left++ + *right++ + *down++) \
47                   / 5.0;
48   }
49   MPI_Barrier(MPI_COMM_WORLD);
50 }
```

Figure 2.7: Parallel version of Jacobi Iterations using MPI.

2.2.2 Remote Memory Access

Remote memory access (RMA) constitute an intermediate programming paradigm between message passing and shared memory. This model combines some advantages of shared memory, such as direct access to shared/global data, and some advantages of message-passing model, such as the knowledge of data locality. In some cases, remote memory operations can be used as a high performance alternative to message passing when RMA is directly supported by the hardware. The SPMD model is often used with RMA, but since symmetry in the communication is not enforced by RMA, other programming models are also used.

MPI

As previously mentioned (Section 2.2.1), MPI version 2.1 supports one-sided communication and thereby RMA. However, the RMA paradigm used in MPI are more restrictive than other dedicated RMA libraries. MPI operates with active target and passive target communication.

Active target is similar to message passing, in which both communicating processes are explicitly involved in the communication. Except that all the data transfer arguments are provided by one process, and the second process only participates in the synchronization.

Passive target is more like normal RMA libraries, in which only one processes provide the data transfer arguments and the synchronization. Both active and passive target support asynchronously communication, but it is not possible to initiate multiple transfers and then wait for one specific transfer to complete. Instead you have to wait for all active transfers to finish. Also it is not possible to lock a subpart of a remote memory window – the entire window must be locked.

These restrictions in MPI are not natural to the RMA paradigm and are not in vendor-specific interfaces such as the Cray SHMEM[9] and IBM LAPI[88]. However, it may help reduce the opportunities for writing erroneous application since synchronization is enforced by some of the restrictions.

ARMCI

Aggregate Remote Memory Copy Interface[78] is a remote memory access communication interface that is designed to be used in libraries rather than directly in user applications. ARMCI leaves the library developer in charge of managing protection and consistency of data accessed by RMA communication and does not support heterogeneous environments. The focus is to deliver performance while providing a widespread portability for homogeneous hardware platforms.

Good performance is demonstrated in [78] where ARMCI is compared with network supported raw Get/Put communication and MPI two-sided communication on InfiniBand Cluster (Figure 2.8). The overhead associated with ARMCI is low and the performance is almost identical with raw communication. ARMCI outperform MPI when

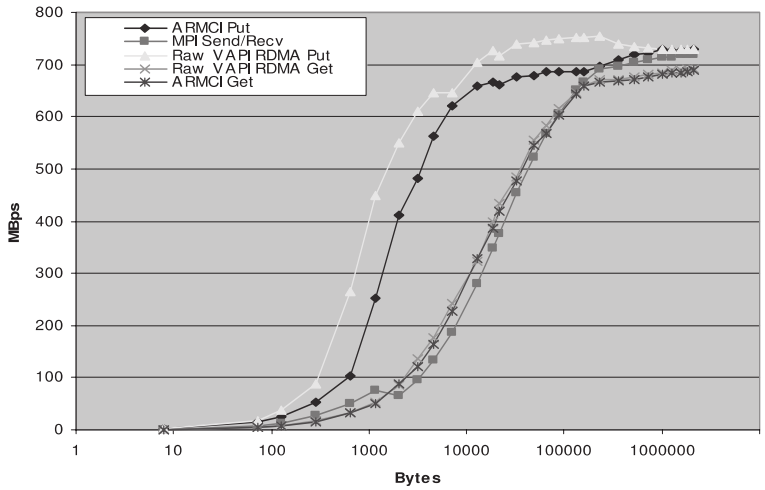


Figure 2.8: Bandwidth in ARMCI Put/Get in comparison to raw Get/Put and MPI Send/Recv (two-sided) on Infiniband Cluster. (From [78]).

transferring relative small messages, but the performance is very similar when transferring large messages.

However, the performance difference may not be that significant when using MPI one-sided communication. [60] demonstrates that it is possible to archive better performance with MPI one-sided communication instead of two-sided communication on Infiniband Cluster. This is accomplished be utilizing InfiniBand RDMA[1] operations, which completely eliminates the involvement of the passive process in one-sided communication.

2.2.3 Libraries and Languages

Libraries and programming languages that support parallelization on distributed memory architectures is a well known concept. The existing tools either seek to provide optimal performance in parallel applications or seek to ease the task of writing parallel applications.

The development of applications that utilize distributed memory architectures often involves distributed operations on distributed data structures. It is therefore the goal for many libraries to directly support such distributed operations and some libraries go even further by seamlessly maintaining the distributed data structures.

In the following, we explore some of the libraries and programming languages that focus on distributed operations and data structures.

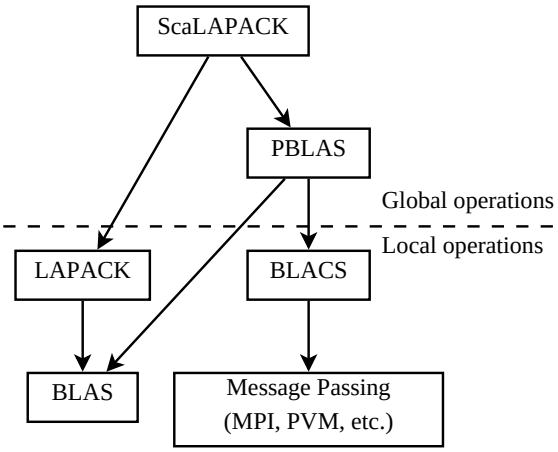


Figure 2.9: ScaLAPACK Software Hierarchy.

ScaLAPACK

The library ScaLAPACK[15] is a parallel library based on LAPACK that targets distributed memory architectures. ScaLAPACK support a subset of the operations available in LAPACK, but instead of working on local vectors and matrices ScaLAPACK works on distributed vectors and matrices.

ScaLAPACK is used together with the SPMD model and users are required to make use of Message Passing Programming to utilize ScaLAPACK. The software hierarchy is quite complex and make use of both distributed and shared memory linear algebra libraries (Figure 2.9).

Parallel BLAS (PBLAS) is a library developed as part of ScaLAPACK and it is used for all distributed BLAS operations. PBLAS makes use of BLAS locally in a process and the communication library Basic Linear Algebra Communication Subprograms (BLACS)[38] is used for all inter-processes communication.

ScaLAPACK and PBLAS makes use of the Two-Dimensional Block Cyclic Distribution scheme, which has a good load balance in numerical problems that have a diagonal computation workflow e.g. Gaussian elimination. The distribution scheme is demonstrated in High Performance Fortran[73]. It works by arranging all processes in a two dimensional grid and then distributing data-blocks in a round-robin fashion either along one or both grid dimensions (Figure 2.10); the result is a well-balanced distribution.

Global Arrays

Global Arrays[79] is a library that provides an efficient and portable shared memory programming interface for distributed memory computers. Global Arrays support both distributed and shared memory architectures but make most sense in a distributed environment.

[1]Remote Direct Memory Access

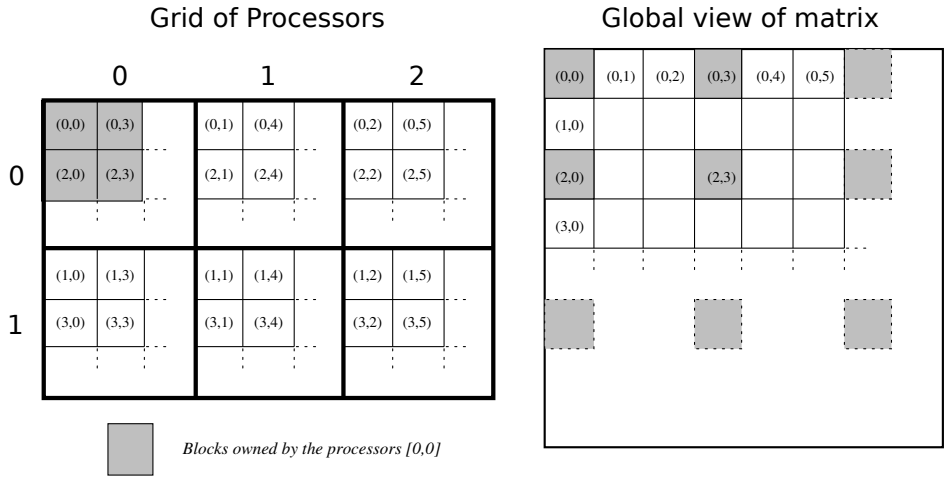


Figure 2.10: The Two-Dimensional Block Cyclic Distribution of a matrix on a 2 x 3 grid of processors.

Global Arrays have been designed to complement rather than substitute for the message passing paradigm. The user can use both the shared memory paradigm and the message passing paradigm in the same program. MPI is therefore often used in combination with Global Arrays.

Central in Global Arrays is a distributed data object, a *global array*, which makes the data distribution transparent to the user. Two categories of operations are support for global arrays. First category is numerical operations that involve whole global arrays, such as Gaussian elimination and matrix multiplication. The operations are used similar to collective operations in MPI, which require the involvement of all MPI processes, except that no process specific arguments is required. Second category is accessing the global arrays from a process. This is much like shared memory programming – a process download a subpart of a global array to a local buffer, manipulate the data locally and then upload the data back to the global array. Global Arrays supports, like most shared memory environments, both shared and exclusive locking of subparts of the global array. When a global array is distributed on multiple cluster nodes, it will have a non-uniform memory access characteristic. Some parts of the global array are local and some parts are remote from a nodes point of view. Global Array exposes this to the user by providing functions that returns coordinates for the local and remote part. However, even when a process only access a local part of a global array that part still needs to be copied to a local buffer. This is one of the main drawbacks with Global Arrays.

In relation to the work with GPAW[2] I have investigated whenever Global Arrays is a better performing alternate than a MPI only implementation. Two finite different implementations – one using Global Arrays and one using MPI – are executed on a Blue Gene/P supercomputer. The result shows that in my test case the MPI implementation performs better than the Global Arrays implementation (Figure 2.11). The main reason

[2]The GPAW project is discussed in section 4

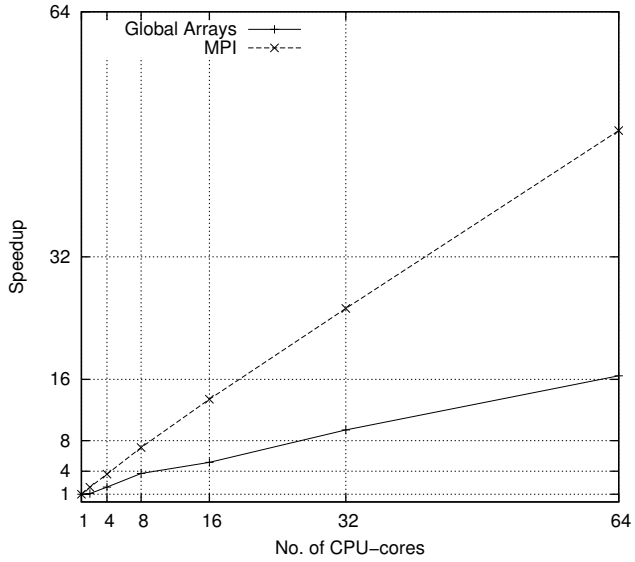


Figure 2.11: Speedup of a finite different computation on a Blue Gene/P supercomputer. Two implementations are compared – one using MPI and one using Global Arrays.

for this performance penalty is the required copying of local data.

IPython

IPython[85] is an interactive shell for the Python programming language that, among other features, support distribute and parallel executing. IPython supports both the Fork/Join and the Message Passing parallel paradigm, but since the focus of IPython is interactive programming the Fork/Join paradigm is most often used. IPython does not seamlessly parallelize applications but instead provides a broad range of useful parallel functions and language directives. Message Passing in IPython is build on MPI and requires MPI binding for Python. The programming model is very similar to ScaLAPACK and support parallel operations on distributed data structures while it is the responsibility of the programmer to ensure the data distribution.

2.2.4 Partitioned Global Address Space Languages

Partitioned Global Address Space (PGAS) [**?**, **?**], also known as distributed shared memory, is a group of languages that are designed around a memory model in which a global address space partitioned such that a portion of it is local to each processor. PGAS languages offer programming abstractions similar to shared memory, but with control over data layout that is critical to high performance and scalability. The abstraction is therefore higher than one-sided and two-sided message passing libraries, but users still program with the SPMD model in mind, writing code with the understanding that multiple instances of it will be executing cooperatively.

Co-array Fortran

Co-Array Fortran[80] is a small extension of Fortran-95 for parallel processing on Distributed Memory Machines. Co-Array Fortran introduces new type of array dimension called a *co-array*. By declaring a variable with a co-array dimension, the user specifies that each process will allocate a copy of the variable. Each process can then access remote instances of the variable by indexing into the co-array dimensions. Co-arrays are expressed using square brackets which make them stand out syntactically from traditional Fortran arrays and array references. Synchronization routines are also provided to coordinate between the cooperating processes.

Unified Parallel C

Unified Parallel C (UPC)[25] is similar to Co-array Fortran in that it extends C to support PGAS-style computation and also introduces a new distributed array. Declaring an array variable with the *shared* keyword causes the array elements to be distributed between the program processes (or threads) in a cyclic or block-cyclic manner. UPC uses the one-sided communication library GASNet[19], which has many similarities with the ARMCI library previously described.

Titanium

Titanium[102] is a language that was developed at Berkeley as an SPMD dialect of Java. Titanium provides a global memory space abstraction whereby all data has a user-controllable processor affinity, but parallel processes may directly reference each other's memory to read and write values. Titanium is essentially a superset of Java 1.4 and inherits all the expressiveness, usability and safety properties of that language. However, the Titanium compiler uses a source-to-source model, translating the Java-dialect into the C programming languages. The communication is than translated into either MPI or GASNet function calls and it is thereby possible to use Titanium on most platforms.

2.2.5 High Productivity Computing Systems

In 2002 an interesting research program were launched by the Defense Advanced Research Projects Agency, which is a U.S Department of Defense institute. The program is called High Productivity Computing Systems (HPCS) and offers funding for industry and academia to research the development of computing systems that focus on high productivity along with high performance. Part of this program concentrates on the specification of novel languages for the HPC community. In Phase 2 of the program (July 2003 – July 2006) the three remaining partners that were awarded funding were Cray, IBM and SUN. SUN were eventually dropped from the program at the start of Phase 3. SGI and HP were part of Phase 1, but did not receive the funding to continue their research.

Three new programming languages was developed through the HPCS program – X10[30] by IBM, Chapel[28] by Cray, and Fortress[5] by SUN. All three languages are PGAS oriented, like Co-array Fortran, UPC, and Titanium, but they do not use the SPMD programming model. Instead they make use of a global programming view in which, from the perspective of the user, only one instance of the application is executed. Still, the user needs to handle parallel programming aspects, such as whenever a variable should be shared between processes or not.

X10

X10 is a type-safe, parallel, distributed object-oriented language intended to be very easily accessible to Java programmers, i.e. the syntax is very similar to Java. X10 targets non-uniform memory hierarchies, which both include single processors, like the Cell Broadband Engine[62], and multiple-processor computer systems.

A computation is divided among a set of *places*, each of which holds some data and hosts one or more activities that operate on those data. The *places* are managed by the user and it is op to the user to distributed the workload between them. X10 introduces a distributed array that spans over multiple *places*. New arrays can be created by combining multiple arrays, performing element-by-element operations on arrays (with the same distribution), and by using collective operations, e.g. scan, that are executed in parallel on existing arrays to create new arrays.

To prevent deadlocks X10 uses the concept of parent and child relationships for activities. An activity may spawn one or more child activities, which may themselves have children. Children cannot wait for a parent to finish, but a parent can wait for a child using the finish command.

Chapel

Chapel is a object-oriented programming language that strives to vastly improve the programmability of large-scale parallel computer. Chapel supports both task and data parallelism – task parallelism uses a multi-threaded execution model in which the user assigns different tasks to different threads. Data parallelism, on the other hand, uses a more global view in which the user operates directly on distributed arrays through global iterators.

Chapel is designed around a multi-resolution philosophy, permitting users to initially write very abstract code and then incrementally add more detail until they are as close to the machine as their needs require.

Fortress

Fortress is a general-purpose, statically typed, programming language that uses a completely different approach than the other languages discussed. The main idea is that it should be possible to use standard mathematical notation directly in the source code.

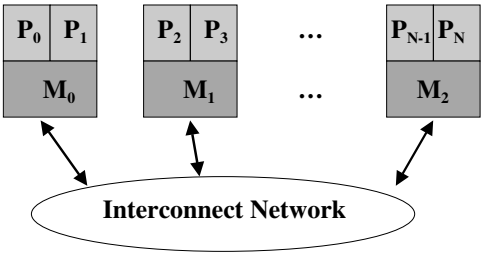


Figure 2.12: Hybrid Memory.

For this purpose, Fortress makes use of Unicode to support a great diversity of mathematical notations. Furthermore, concepts that are important for scientific computation, such as vectors, matrices, and physical units (meters, seconds, etc) are supported.

Unlike most parallel languages, Fortress is implicitly parallel wherever possible and provides constructs and annotations to serialize execution when necessary. As a result, a compiler or virtual execution environment need not concern itself with determining whether executing a program in parallel is permissible, only whether doing so is advantageous. To avoid typical race conditions, Fortress supports transactional memory through the use of atomic expressions.

2.2.6 Incorporate Parallelism into Existing Languages

Instead of designing new parallel languages or libraries, some projects incorporate parallelism into existing sequential languages. This is typically done by translating the sequential source code into a parallel program or by replacing the data backend with a distributed data structure.

In [45] the authors have developed a framework that makes it possible to utilize GPUs in Python+NumPy applications. The application is compiled into C++, and NumPy vector operations and annotated functions are translated into GPU kernels. The GPU utilization is almost completely transparent, still the user is required to specify which Python/NumPy functions that should make use of the GPU. Furthermore, the user has to annotate the variable types used in the Python application.

Another approach is used in the parallel MATLAB framework MATLAB*P[32], which replace the standard array backend with a new distributed array backend. All MATLAB operations that include this new array will seamlessly execute in parallel. The approach is therefore very non-intrusive to the existing sequential code and only minor work is required by user to utilize distributed memory architectures.

2.3 Combining Distributed and Shared Memory

The combination of Distributed and Shared Memory programming is called Hybrid programming and is becoming more relevant with the increase in popularity of SMP architectures. Because a strong trend in HPC hardware is towards systems of shared-memory

computation nodes, users will have to deal with parallelism both on the SMP level and on the inter-node level. Thus the Hybrid programming approach may have an advantage over strictly Distributed or Shared Memory programming.

The idea is to exploit the memory locality on the SMP nodes and thereby avoid communication between CPU-cores on the same node. Unfortunately, it is not trivial to obtain good performance when combining shared memory programming with distributed memory programming. Even though some internal communication is avoided, it is often the case that the sole use of distributed memory programming outperforms a combination of threads and Message Parsing when computing on clusters of SMP nodes[54]. In [56] the authors demonstrate that it is possible to obtain identical performance with MPI and OpenMP compared with pure MPI implementation.

The authors in [24] put forth some observations about the hardware and software that influence the performance of hybrid programming. One observation is that on a well balanced system, a loop level parallelization approach is unfavorably compared to a strictly MPI implementation. Another observation is that a system of SMP nodes performs better with a pure MPI approach for latency sensitive programs and worse for bandwidth sensitive programs. Thus under the right circumstance it is possible to obtain good performance with hybrid programming.

2.4 Vector Oriented Programming

Vector-oriented programming introduces an abstraction level that hides the parallelism altogether. The idea in vector-oriented programming (also known as array programming) is to express algorithms using high-level array operations, e.g. by expressing the addition of two arrays using one high-level function instead of computing each element individually.

The programming model uses a single thread execution model that only uses parallelization implicitly. It is well suited for expressing data-centric parallelism and avoids many nasty parallel programming bugs, such as deadlocks and data races.

2.4.1 High Performance Fortran

High Performance Fortran (HPF) [73] is a extension of the Fortran-90[3] programming language, which consist of new directives to specify data dependencies and data distribution schemes. It is up to the HPF compiler to parallelize the application and distribute the used data objects – no explicit communication is required. However, in order to obtain good performance HPF dependence on the vector-oriented programming model where the user *align* arrays together to reduce communication.

HPF supports vector-oriented programming by the exploitation of the array operations introduced in Fortran-90. For instance the statement $A = B + C$, where all three variables are arrays, means add B and C together element-wise and store the result in A.

2.4.2 Z-level Programming Language

Z-level Programming Language (ZPL) [29] uses an array abstraction to implement the vector-oriented programming model. ZPL has no parallel directives or other forms of explicit parallelism instead ZPL uses data parallelism to execute array operation in parallel. Rather than machine code, the ZPL compiler translates ZPL code into ANSI C code that makes use of the MPI library for communication. Therefore, ZPL is fairly portable – in order to support a platform only an ANSI C compiler and a MPI library must be available.

Chapter 3

Target Architectures

In order to obtain good scalable performance, it is crucial to optimize the utilization of the underlying hardware. In this work, the main architectural focus is Distributed Memory Machines, which is a collection of interconnected computer nodes (Figure 2.5). Each node has its own private memory but a node may consist of multiple CPU-cores that share the memory. The configurations of such installations are endless – from clusters of personal computers, Beowulf Clusters, to specialized supercomputers – yet the machine must be considered as one installation.

In this section I will describe two of the fastest supercomputers in the world – one of which I got the chance to gain firsthand experience in working with. But first a brief description of the heart in any Distributed Memory Machine: the network.

3.1 Network

There is a great diversity in the networks that are being used in Distributed Memory Machines. As of 2010, the three most popular networks on the Top500 list[75] are Gigabit Ethernet, Infiniband[89], and Myrinet[18] with a share of 51.8%, 36.2%, and 1.4%, respectively. Common for all three networks is that they are fairly generic and

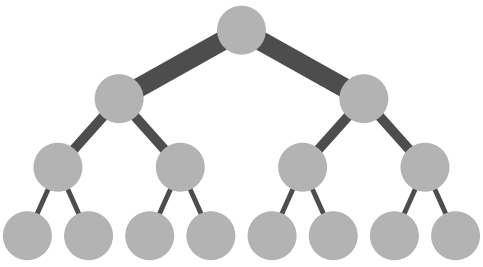


Figure 3.1: Fat-Tree Network Topology: a very popular topology used in Distributed Memory Machines, in which the links become *fatter* as one moves up the tree towards the root [72].

can be used in most Distributed Memory Machine designs. Typically, the network is connected using a number of switches in a tree-structured topology (Figure 3.1). This is opposed to networks that are more non-generic, in which switches and the network topology are incorporated directly into the machine architecture.

Gigabit Ethernet is the most popular network technology, probably because it is the cheapest. But with communication latencies in the order of 25 μs [**?**] the applicability of Gigabit Ethernet is too restricted to be the ideal network. For tightly coupled parallelism both Infiniband and Myrinet are superior with communication latencies around 1-3 μs [**?**]. 10 Gigabit Ethernet is closing the gab to Infiniband and Myrinet with communication latencies around 2-4 μs [**?**].

The Top500 list is based on the LINPACK Benchmark[37] which again is based on tightly coupled parallelism. This is evident when examining the relationship between the peak performance and the sustained performance of machines on the Top500 list (2010). Machines that uses Gigabit Ethernet only utilize 50% of the peak performance whereas machines that uses Infiniband and Myrinet utilize 77% and 72%, respectively.

3.2 Roadrunner

Roadrunner[10] is a hybrid-architecture supercomputer developed by Los Alamos National Laboratory and IBM. It contains 12,240 IBM PowerXCell 8i processors and 6,480 AMD Opteron cores in 3,060 compute nodes. The Roadrunner is one of the fastest supercomputer in the world with a sustained maximum performance of 1.026 PFlops. It was the first machine to exceed the 1 PFlops barrier, which happened in May 2008.

Roadrunner exposes a diverse computational environment given its heterogeneity. It can be utilized in one of three main programming paradigms depending on the suitability of each application. An application can run unmodified using only the Opteron processors without acceleration by the PowerXCell 8i processors. Or the application can use both processor types, accelerating key performance hotspots of the code on the PowerXCell 8i without porting all of the code. Or the application can run on the PowerXCell 8i processors for all computational tasks and employ the Opterons only as support for internode communication, I/O, and visualization.

Approximately 95% of the peak performance of Roadrunner results from the PowerXCell 8i processors, which are used as accelerators for the AMD Opteron processors. PowerXCell 8i is an improved version of the Cell Broadband Engine (Cell BE)[62], which was design for the Sony PlayStation3. A single Cell BE has a peak performance of 217.6 GFlops from its nine processor-cores. However, this speed is limited to single-precision operations and drops to 21.0 GFlops for double-precision. A further limitation is the use of a low-performance PowerPC processor core, which typically achieves a quarter of the performance of a typical AMD Opteron core. Finally, the memory controller supports only Rambus XDR memory, limiting memory capacity to 2GB.

To overcome these limitations, IBM implemented a new Cell processor, the PowerXCell 8i, for use in Roadrunner. The PowerXCell 8i has a peak performance of 108.8 Gflops/s on double-precision operations, and supports DDR2 memory at 800MHz, al-

lowing up to 32GB memory. The remaining shortcoming, the low-performance PowerPC processor core, is overcome by the incorporation of dual-core AMD Opteron 2210 HE processors with one Opteron core for each PowerXCell 8i processor. In effect this provides an accelerator to each Opteron core.

3.2.1 The node design

A Roadrunner compute node is built using a triblade configuration (Figure 3.2). One blade, an IBM LS21, contains two dual-core Opteron processors, and the remaining two blades, IBM QS22s, each contain two PowerXCell 8i processors. Each Opteron core and PowerXCell 8i within the triblade has 4 GB of DDR2 memory. The Opteron processors are clocked at 1.8 GHz, with a peak performance of peak of 14.4 GFlop per LS21 blade. Each core has a 64 KB L1 data cache, a 64 KB L1 instruction cache, and a 2 MB L2 cache.

The PowerXCell 8i processors are clocked at 3.2 GHz, and contain one Power Processing Element (PPE), and eight Synergistic Processing Elements (SPEs). The PPE has a traditional cache-based memory hierarchy consisting of a 32 KB L1 data cache, a 32KB L1 instruction cache, and a 512 KB L2 cache. A key characteristic of the SPE is that it can directly address only 256 KB of memory; this high-speed memory, known as local store, takes the place of a conventional cache architecture. Main memory, shared with the PPE, can be accessed only via explicit direct memory access (DMA) transfers to or from local store.

3.2.2 Network

From the perspective of the user, Roadrunner is seen as a network of dual-core Opteron processors, in which each Opteron core has a Cell processor with eight SPEs attached (Figure 3.3). The Opteron network is utilized like conventional Distributed Memory Machines, e.g. by using MPI, and the SPEs are seen as local accelerators. Roadrunner uses an InfiniBand network to interconnect all 3,060 compute nodes (triblades). The topology is a Fat-Tree (Figure 3.1) with at most seven hops between two nodes and an average of 5.38 hops.

The latency between two Opteron processors on different nodes is approximately 2.16 μs, which is similar to other modern InfiniBand networks. However, the latency between two SPEs on different nodes is a much higher 8.78 μs. The performance bottleneck is the internal PCI Express channel between the SPE and the Opteron processor, which has a communication latency of 3.19μs (Figure 3.4). In spite of this, both [65] and [10] demonstrates good system-wide speedup. A full system execution of the LINPACK benchmark results in a 74.6% utilization of the peak performance (Figure 3.5).

3.3 Blue Gene/P

The Blue Gene/P (BGP)[95, 91] is a supercomputer developed by IBM and is the successor to the Blue Gene/L. The full 72 racks BGP installation at JUGENE is, as of 2010,

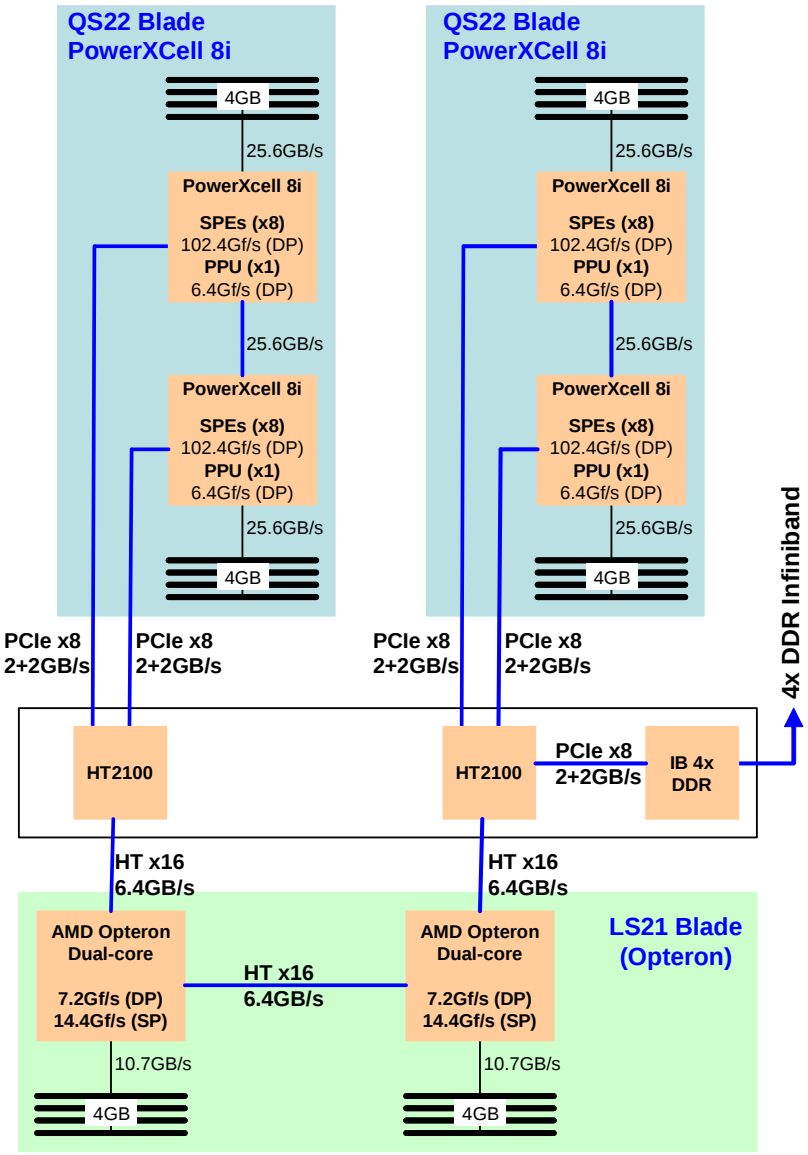


Figure 3.2: A Roadrunner Triblade (From [10]).

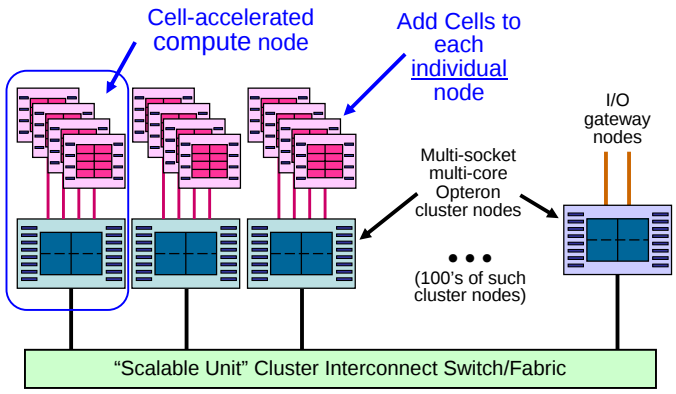


Figure 3.3: Roadrunner network seen from the perspective of the user.

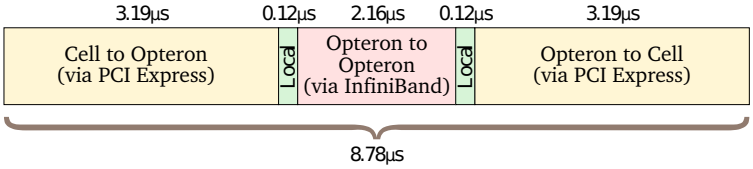


Figure 3.4: Breakdown of the latency of a zero-byte message as it travels from a Cell to another Cell located in a different node (From [10]).

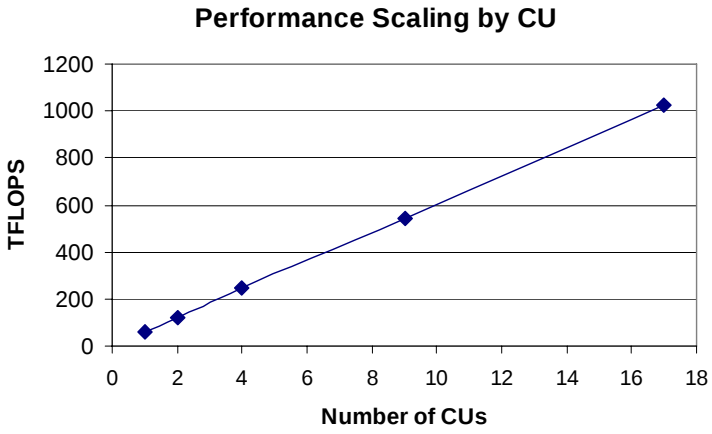


Figure 3.5: Speedup of the LINPARCK benchmark – going from 180 to 3,060 compute nodes. **CU** is a node group that consist of 180 nodes (From [65]).

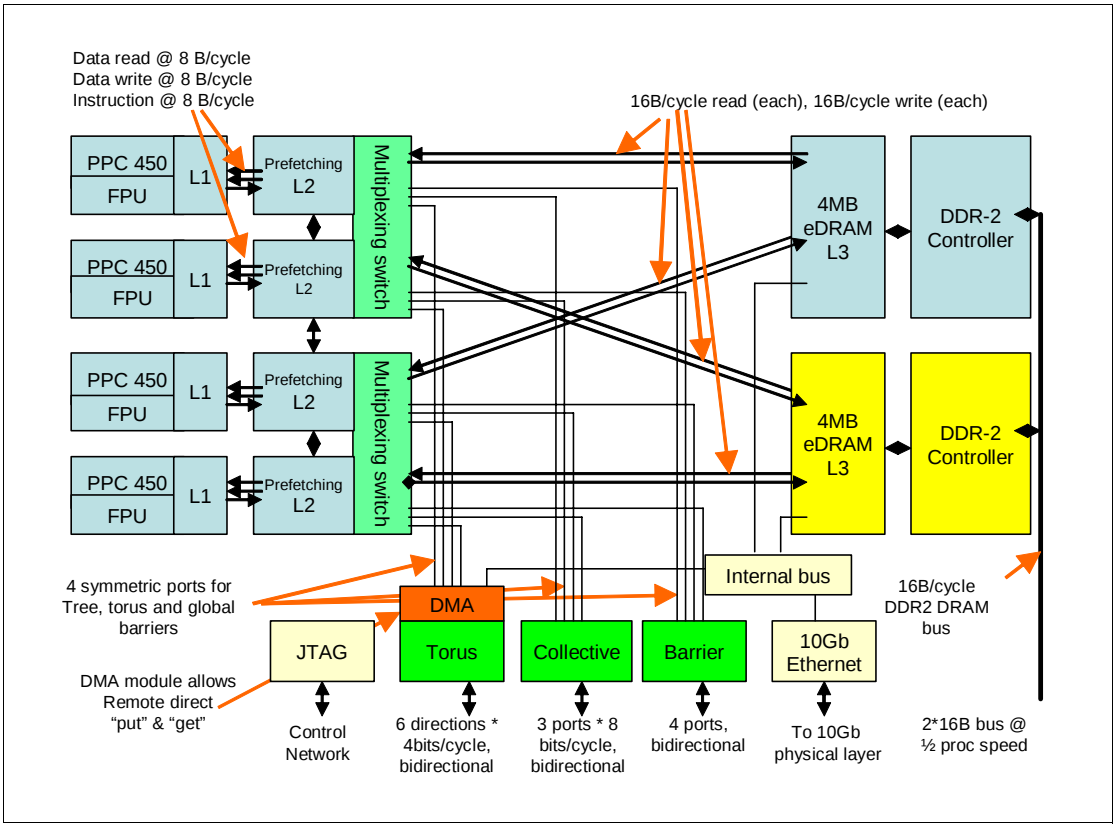


Figure 3.6: A Blue Gene/P Node (From [91]).

the fourth fastest supercomputer in the world with a sustained maximum performance of 807 TFlops.

The design philosophy behind the BGP is based on the observation that the von Neumann bottleneck associated with main memory and network is the greatest challenge when scaling to the extreme. To sustain good scalability it is important that the ratio between the speed of memory, communication and processor are balanced. BGP accomplish this balance by combining a relatively low clocked CPU with a modest performing main memory and an extremely efficient communication system.

3.3.1 The node design

A Blue Gene/P node contains four 850MHz PowerPC 450 processors with 2 GB of shared SDRAM-DDR2 and three levels of cache – A private level one and two cache, and a 8 MB shared level three cache (Figure 3.6). The main memory bandwidth is 16 GB/s and peak performance of a node is 13.6 Gflops.

A node is implemented as a system-on-a-chip and consumes only 33 watts, which makes it possible to have 1024 nodes in one rack.

3.3.2 Network

Blue Gene/P consists of no less than five independent networks: a 3D torus network, a collective tree structured network, a global barrier network, a dedicated 10 gigabit Ethernet network for I/O, and a 1 gigabit Ethernet network for administration. In short, the performance of the network is quite impressive – the following is a brief description of the three networks that are explicitly used by the user.

Three-dimensional torus The torus network is used for general-purpose, point-to-point message passing. The topology is a three-dimensional torus constructed with point-to-point, serial links between routers embedded within the BGP nodes. Therefore, each node has six nearest-neighbor connections. The target hardware bandwidth for each torus link is 425 MB/s in each direction of the link for a total of 5.1 GB/s bidirectional bandwidth. Hardware latency for the nearest neighbor is an impressive 100 ns (32-byte packet) and 800 ns (256-byte packet). To offload torus communication from the CPUs a node is equipped with a direct memory access (DMA) engine.

Global collective The global collective network is a high-bandwidth, one-to-all network used for collective communication operations, such as broadcast and reductions. Each node has three links to the global collective network at 850 MB/s per direction for a total of 5.1 GB/s bidirectional bandwidth per node. Latency on the global collective network is less than 2 μs from the bottom to top of the collective, with an additional 2 μs latency to broadcast to all.

Global barriers The global interrupt network enables fast signaling of global barriers. Round-trip latency to perform a global barrier over this network for a full 73728 node partition is approximately 1.3 microseconds.

To investigate the overhead associated with the software layer and how much the message size influence point-to-point bandwidth, I have performed an experiment in which one MPI message is send between two neighboring BGP nodes (Figure 3.7). The result of the experiment clearly shows that in order to maximize the bandwidth, a message size greater than 10^5 bytes is needed, while half the asymptotic bandwidth is achieved at approximate 10^3 bytes.

3.3.3 Application Development

The Blue Gene/P supports shared memory programming with pthread or OpenMP, but with the limitation that BGP only supports one thread per CPU-core. However, it is also possible to utilize the four cores on a node by use of the *virtual partition mode*, which is a virtual partition of node supported by BGP. From the programmers point of view the four CPU-cores would then look like four individual nodes with each 512MB of main memory. This virtual partitioning is called virtual mode.

BGP also implements the MPICH2[49] library which comply with the MPI-2 specification. MPI-2 specifies different levels of threaded communication. BGP supports the

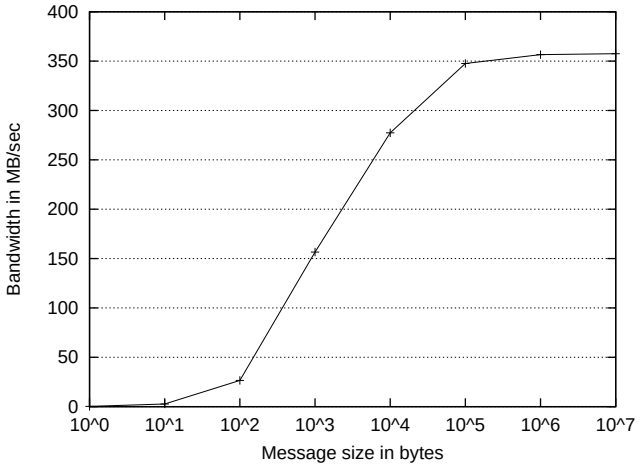


Figure 3.7: A bandwidth graph showing how the message size influence the bandwidth. In this experiment, one MPI message is send between two neighboring BGP nodes.

fully thread-safe mode called `MULTIPLE`, which allows any thread to call the MPI library at any time. Since there is an overhead associated with `MULTIPLE` (e.g. locks), it is also possible to use the more restricted `SINGLE` mode, which do not allow concurrent calls to MPI.

The MPICH2 implementation is tailored to utilize the BGP's DMA engine, which means that non-blocking MPI communication is handled asynchronously with minimum CPU involvement.

BGP supports the `MPI_Cart_create` function which tells BGP to reorder the MPI ranks in order to match the torus network.

3.3.4 Argonne National Laboratory

In my involvement with GPAW (Section 4) I was granted access to the Blue Gene/P installation at Argonne National Laboratory (ANL) in Chicago. The installation consists of 40 racks and is one of the fastest supercomputer in the world with a sustained maximum performance of 448 TFlops. However, only four racks, which translates to 16384 CPU-cores, was accessible for the GPAW project.

Error in the communication layer

In my intense work with the MPICH implementation on the Blue Gene/P, I encountered a strange runtime error. The error message was this:

```
mpid_rma_common.c:1520:
recv_sm_cb: Assertion ‘(win)->_dev.epoch_rma_ok
&& !((win)->_dev.epoch_assert & 4096)’ failed.
Abort(1) on node 2 (rank 2 in comm 1140850688): Fatal error in
MPI_Win_unlock: Wrong synchronization of RMA calls , error stack:
MPI_Win_unlock(116).: MPI_Win_unlock(rank=3, win=0xa0000000) failed
```

```
MPID_Win_unlock(861): Wrong synchronization of RMA calls
```

At first I thought that the error should be found in my implementation, but after a lot of debugging I came up with a valid piece of one-sided MPI communication that sometimes fails:

```
rank = (myrank+1)%worldsize
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, rank...)
MPI_Put(..., rank...)
MPI_Win_unlock(rank...)
```

I filed a bug report, which was first handled by the Blue Gene team at ANL, then by the MPICH team also at ANL, and ended at the IBM's Blue Gene/P team. The result was two official patches to the Blue Gene/P software stack:

```
patch0 (Douglas Miller, IBM): fixes the perfromance issue for
                              one-sided MPI_Put/MPI_Get.
patch1 (Douglas Miller, IBM): fixes the race condition in MPI_Win_unlock.
```

This story goes to show that HPC pushes the envelope for the possibilities of hardware architectures and is for experts.

Chapter 4

Scientific Application: GPAW

As part of this thesis, I got the opportunity to gain firsthand knowledge in the process of parallelizing a scientific application for a supercomputer. The idea was that through the involvement I would gain insights into common parallel constructs used in scientific applications and how to improve the scalability of such constructs.

The application, Grid Based Projector Augmented Wave (GPAW)[76], is a simulation software, which simulates many-body systems at the sub-atomic level. GPAW is primarily used by physicists and chemists to investigate the electronic structure, principally the ground state, of many-body systems.

GPAW is mainly developed by a team at the Technical University of Denmark and I had the privilege to work with them on optimizing GPAW for the Blue Gene/P architecture. The focus of my work was to optimize a distributed stencil operation that makeup a substantial part of the overall execution time.

4.1 Introduction

The current trend in HPC hardware is towards systems of shared-memory computation nodes. The Blue Gene/P also follows this trend and consists of four CPU-cores per node. Furthermore, it is quite possible that future versions of the Blue Gene architecture will consists of even more CPU-cores per node.

To exploit the memory locality in the nodes of the Blue Gene/P a paradigm, which combines shared and distributed memory programming may be of interest. We evaluate two different hybrid programming approaches. One approach in which inter-node communication is handled individually by every thread and another approach in which one thread handles the inter-node communication on behalf of all the other threads in a node. The work shows that, on the Blue Gene/P, the first approach is clearly superior the latter. In [24] the authors concludes that, on a well balanced system, a loop level parallelization approach, corresponding to our second hybrid approach, is unfavorably compared to a strictly MPI implementation. Our first hybrid approach was developed on the basis of that conclusion.

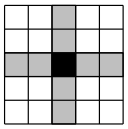


Figure 4.1: A stencil operation on a 2D grid.

4.2 GPAW

GPAW is a real-space grid implementation of the projector augmented wave method[16]. It uses uniform real-space grids and the finite-difference approximation for the density functional theory calculations.

A central part of density functional theory and a very time consuming task in GPAW, is to solve Poisson and Kohn-Sham equations [**?**, **?**]. Both equations rely on stencil operations when solved by GPAW. When solving the Poisson equation, a stencil is applied to the electrostatic potential of the system. When solving the Kohn-Sham equation, a stencil is applied to all wave-functions in the system. Both the electron density and the wave-functions are represented by real-space grids. A system typically consists of one electron density and thousands of wave-functions. The number of wave-functions in a system depends on the number of valence electrons in the system. For every valence electron there may be up to two wave-functions.

The computational magnitude of a GPAW simulation depends mainly on three factors: The world size, simulation system resolution and the number of valence electrons. The world size and resolution determine the dimensions of the real-space grids and the number of valence electrons determines the number of real-space grids.

A user is typically more interested in adding valence electrons to the simulation than to increase the size or resolution of the world. The real-space grid size will ordinary be between 100^3 to 200^3 where as the total number of real-space grids will be greater than one thousand.

4.2.1 Stencil Operation

A stencil operation updates a point in a grid based on the surrounding points. A typical 2D example is illustrated in Figure 4.1 where points are updated based on the two nearest points in all four directions.

Stencil operations on the real-space grids (3D arrays) are used for the finite-difference approximation in GPAW. The stencil operation used is a linear combination of a point's two nearest neighbors in all six directions and itself. The stencil operations do normally use periodic boundary conditions but that is not always the case.

If we look at the real-space grid A and a predefined list of constants C, a point $A_{x,y,z}$

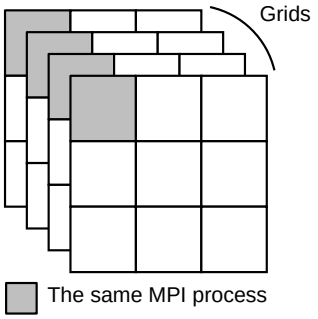


Figure 4.2: Four 2D grids distributed over nine processes.

is computed like this:

$$\begin{aligned} A'_{x,y,z} &= C_1 A_{x,y,z} + C_2 A_{x-1,y,z} + C_3 A_{x+1,y,z} + \\ &\quad C_4 A_{x-2,y,z} + C_5 A_{x+2,y,z} + C_6 A_{x,y-1,z} + \\ &\quad C_7 A_{x,y+1,z} + C_8 A_{x,y-2,z} + C_9 A_{x,y+2,z} + \\ &\quad C_{10} A_{x,y,z-1} + C_{11} A_{x,y,z+1} + \\ &\quad C_{12} A_{x,y,z-2} + C_{13} A_{x,y,z+2} \end{aligned}$$

4.3 The implementation

GPAW is implemented using C and Python. The intention is that the users of GPAW should write the model description in Python and then call C and Fortran functions from within Python. It is in this context a user would apply the C implemented stencil operation on one or more real-space grids.

The parallel version of GPAW uses MPI in a flat programming model and the parallelization is done by simple domain decomposition of every real-space grid in the simulation. That is, every MPI process gets the same subset of *every* real-space grid in the simulation. This is important because some part of the GPAW computation, like the orthogonalization of wave-functions, requires the same subset of every real-space grid in the simulation. This is illustrated in Figure 4.2 with 2D real-space grids instead of 3D grids.

The grids are simply divided into a number of quadrilaterals matching the number of available MPI processes. If no user-defined domain decomposition is present, GPAW will try to minimize the aggregated surface of the quadrilaterals. A real-space grid is represented as a three dimensional array where every point in the grid can be a real or complex number (8 or 16 bytes)

4.3.1 Distributed Stencil Operation

Generally, it should be easy to obtain good scalability for a distributed stencil operation since computation grows faster than communication. If we look at a 3D grid of size $n \times n \times n$ the aggregated computation is $O\left(n^3\right)$ where as the aggregated communication

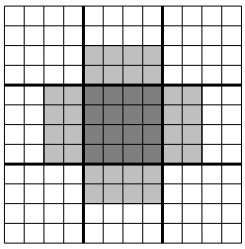


Figure 4.3: 2D grid distributed over nine processes. A process needs some of its neighbor's surface points, to compute its own surface points.

is only $O\left(n^2\right)$. The operation should scale very well when n grows at the same rate as the number of CPUs.

In GPAW, however, scalability is very hard to obtain since the grid size will ordinarily not exceed 200^3. Furthermore, since GPAW requires that every MPI process gets the same subset of every grid, it is hard to take advantage of the fact that the number of grids grows at the same pace as the CPUs.

One feature in GPAW, which makes it easier to parallelize, is the fact that the input grid and the output grid used in the stencil operation is always two separate grids. We need therefore not consider the order in which the grid-points are computed.

Applying a stencil operation on a grid involves all MPI processes. It is possible for an MPI process to compute most of the points in the sub-grid assigned to it. However, points near the surface of the sub-grid, *surface points*, are dependent on remote points located in neighboring MPI processes. This dependency is illustrated in Figure 4.3.

The straightforward approach, and the one used in GPAW, for making remote points available, is to exchange the surface points between neighboring MPI processes before applying the stencil operation. The serialized communication pattern looks like this:

1. Exchange surface points in the first dimension.
2. Exchange surface points in the second dimension.
3. Exchange surface points in the third dimension.
4. Apply the stencil operation.

4.4 Optimizations

In order to make GPAW run faster on the Blue Gene/P, we have explored different optimizations. Optimizations, which have been beneficial, will be discussed in this section.

The most obvious optimization is to exchange surface elements simultaneously in all three dimensions by using the following non-blocking communication pattern:

1. Initiate the exchange of surface points in all three dimensions.
2. Wait for all exchanges to finish.

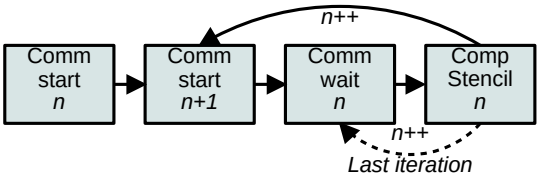


Figure 4.4: Flow diagram illustrating double buffering. The n'th iteration is expressed with a n and Comm and Comp stands for communication and computation, respectively. n++ is an iteration to n's successor.

3. Apply the stencil operation.

The idea is to fully utilize the torus network in all six directions simultaneously.

Another important performance aspect is how to map the distributed real-space grids onto the physical network topology. The 3D torus network is used for point-to-point communication in MPI. Therefore, we will map the distributed real-space grids onto the 3D torus network. Since the grids have the same number of dimensions as the torus network, and since the stencil operation may use periodic boundary condition, a torus topology is a perfect match to our problem. However, the Blue Gene/P requires a partition with 512 or more nodes to form a torus topology. A partition of less than 512 nodes can only form a mesh topology.

4.4.1 Multiple real-space grids

Double buffering and communication batching are two techniques that can improve the performance of the stencil operation. Both techniques require multiple real-space grids but the stencil operation is typically applied on thousands of real-space grids.

Double buffering

Double buffering is a common technique that makes it possible to overlap communication and computation. The following communication pattern illustrates how (Figure 4.4):

1. Initiate the exchange of surface points in all three dimensions for the first grid.
2. Initiate the exchange of surface points in all three dimensions for the second grid.
3. Wait for all exchanges of the first grid to finish.
4. Apply the stencil operation on the first grid.
5. Initiate the exchange of surface points in all three dimensions for the third grid.
6. Wait for all exchanges of the second grid to finish.

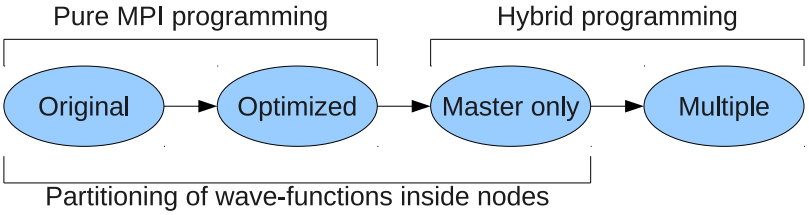


Figure 4.5: A illustrates of the connection between four programming approaches.

The performance gain is dependent on the ability of the MPI library and the underlying hardware to process non-blocking send and receive calls. On the Blue Gene/P, progress in non-blocking send and receive calls will be maintained by the DMA engine and increased performance is therefore expected.

Batching

A method for ensuring critical packet size is to pack real-space grids into batches; inspired by the message size experiment (Figure 3.7).

Continuously dividing the grids between more and more MPI processes reduces the number of surface points in a single sub-grid. That is, at some point the amount of data send by a single MPI call will be reduced to a size in which the MPI overhead and network latency will dominate the communication overhead. The idea is to send a batch of surface points in each MPI call, instead of sending surface points, individually. This will reduce the communication overhead considerably, as the size of the sub-grids decreases. The number of grids packed together in this way, we call *batch-size*.

When using double buffering, it is important to allow the CPUs to start computing as soon as possible. Combining a large batch-size with double buffering will therefore introduce a penalty as the initial surface points exchange cannot be hidden. One approach to minimize this penalty, is to increase the batch-size continuously in the initial stage. For instance a batch-size of 128 could be reduced to 64 in the initial exchange.

4.5 Programming approaches

Different approaches exist when combining threads and MPI. To preserve control we have chosen to handle the threading manually in pthread.

The following is a description of different programming approaches that we have investigated. Every programming approach except the **Flat original** uses the optimizations described in section 4.4.

- **Flat original** is the approach originally used in GPAW. It uses the virtual mode in Blue Gene/P, in which the four CPU-cores are treated as individual nodes, to utilize all four CPU-cores and it is therefore not necessary to modify anything to support the Blue Gene/P architecture.

- **Flat optimized** is an optimized version of the original approach and just like the **Flat original** it uses the virtual mode.
- **Hybrid multiple** does not use the virtual mode. Instead, one hardware thread per CPU-core is spawned. Every thread handles its own inter-node communication. The node will distribute the real-space grids between its four CPU-cores, not by dividing the grids into smaller pieces but by assigning different grids to every CPU-core. Because of this no synchronization is needed until all grids are computed and the synchronization penalty is therefore constant. This way of exploiting multiple grids is the main advantage of this approach.
- **Hybrid master-only** also spawns one thread per CPU-core, but only one thread, the *master thread*, handles inter-node communication. Since we have to synchronize between every grid-computation, each grid-computation will be divided between the four CPU-cores. The synchronization penalty thus become proportional to the number of grids. On the other hand, this approach does work in `SINGLE` MPI-mode and the overhead associated with `MULTIPLE` is therefore avoided.

Figure 4.5 illustrates the connection between the four programming approaches – from the original approach, in which pure MPI programming is used and the wave-functions are partitioned inside the nodes, to the hybrid approach where hybrid programming is used and the wave-functions are shared inside the nodes.

4.6 Results

A benchmark of each implementation has been executed on the Blue Gene/P. 16384 CPU-cores or 4096 nodes or 4 racks were made available to us. Every benchmark graph compares the different programming approaches of the stencil operation in GPAW and a periodic boundary condition is used in all cases.

Figure 4.6 is a classic speedup graph comparing every implemented approach with a sequential execution. It is a relatively small job containing only 32 real-space grids. But because of the memory demand, it is not possible to have more than 32 grids running on a single CPU-core.

The result clearly show that the best scaling and running time is obtained with **Flat optimized** and **Hybrid multiple** both using a batch-size of 8 grids. Since the job only consists of 32 grids a batch-size of 8 is the maximum if all four CPU-cores should be used. Another interesting observation is that the advantage of batching is greater in **Hybrid multiple** than in **Flat optimized**. This indicates that if a job consist of more grids, the **Hybrid multiple** approach may become faster than **Flat optimized**.

4.6.1 Communication and Computation Profile

The communication and computation profile becomes very important when scaling to a massive number of processes. As the number of MPI processes increases the communication time has a tendency to increase due to network congestion. It is therefore essential

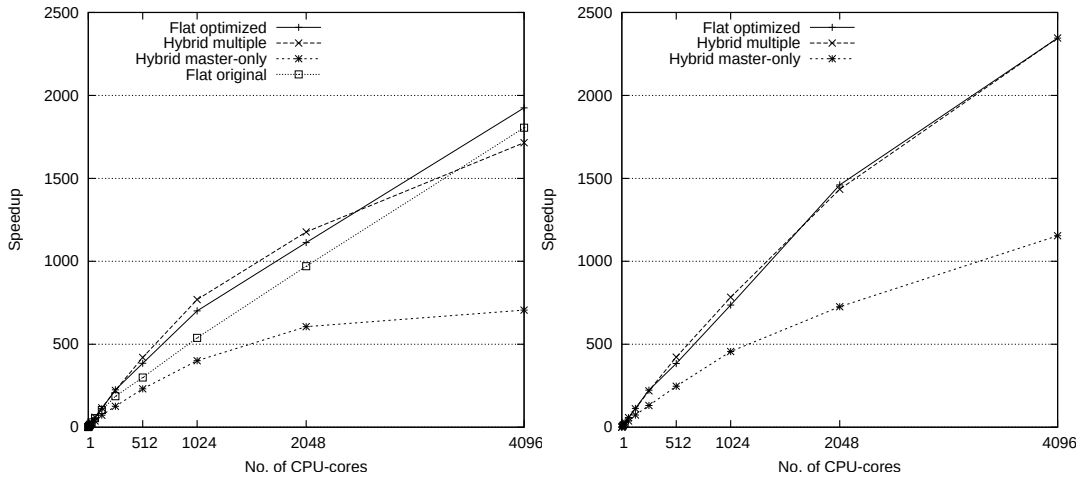


Figure 4.6: Speedup of the stencil operation. The job consist of only 32 real-space grids all with a size of 144^3. In the left graph batching is disabled and in the right graph batching is enabled using a batch-size of 8.

that all communication is spread evenly between the CPU-core and that the diversity of the communication and computation time is minimized.

Figure 4.7 is a profile of the **Hybrid multiple** approach executing on 1024 CPU-cores. It shows a distinct pattern in which the communication and the computation phase are aligned throughout the execution. From that it is evident that **Hybrid multiple** actually do execute in a fairly synchronized manner and no ripple effect of waiting processes is observed.

4.6.2 Multiple real-space grids

As the number of grids grow there is a corresponding linear growth in the computation required in the stencil operation. It is therefore possible to create a Gustafson graph by increasing the number of grids in the same rate as the number of CPU-cores (Figure 4.8). It is important to note that the required communication per node increases faster than the needed computation. This is due to the increased surface size associated with the additional partitioning of the grids. To illustrate this communication increase, the right graph in Figure 4.8 shows the needed communication per node for **Flat optimized** and **Hybrid multiple** respectively.

If we, for example, look at a computation of a grid with a size of 192^3 using 1024 nodes, the grid will either be divided between 1024 MPI processes when using **Hybrid multiple** or 4096 MPI process when using **Flat optimized**. **Flat optimized** needs to communicate approximately 140KB more data per node than **Hybrid multiple**. Note that this is only for a single real-space grid, the difference will grow linearly with the number of grids in the computation.

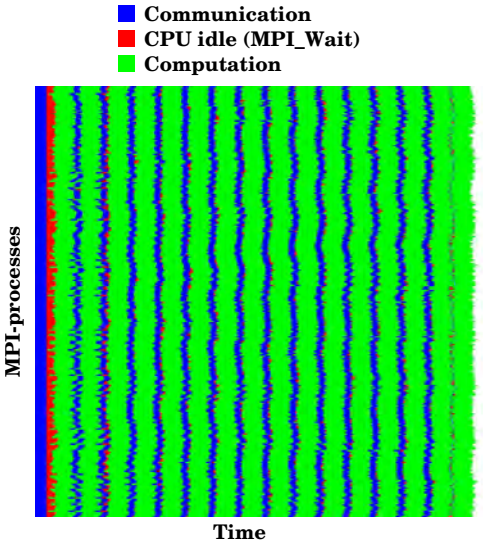


Figure 4.7: Profile of the communication and computation pattern when computing 1024 real-space grids on 1024 CPU-cores and the **Hybrid multiple** approach is used. A line represents a MPI-process and the length of the line represents the progress of time.

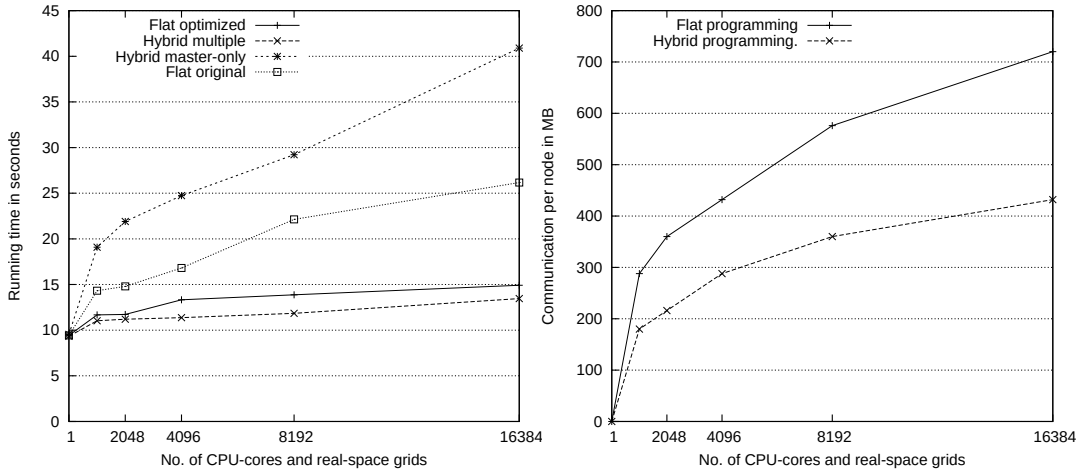


Figure 4.8: Gustafson graphs showing the running time of the stencil operation and the needed inter-node communication when the number of real-space grids is increasing in the same rate as the number of CPU-cores – one grid per CPU-core. The left graph shows the running time and the right graph shows the needed inter-node communication. The grid size are 192^3 and the best batch-size has been found for every number of CPU-cores.

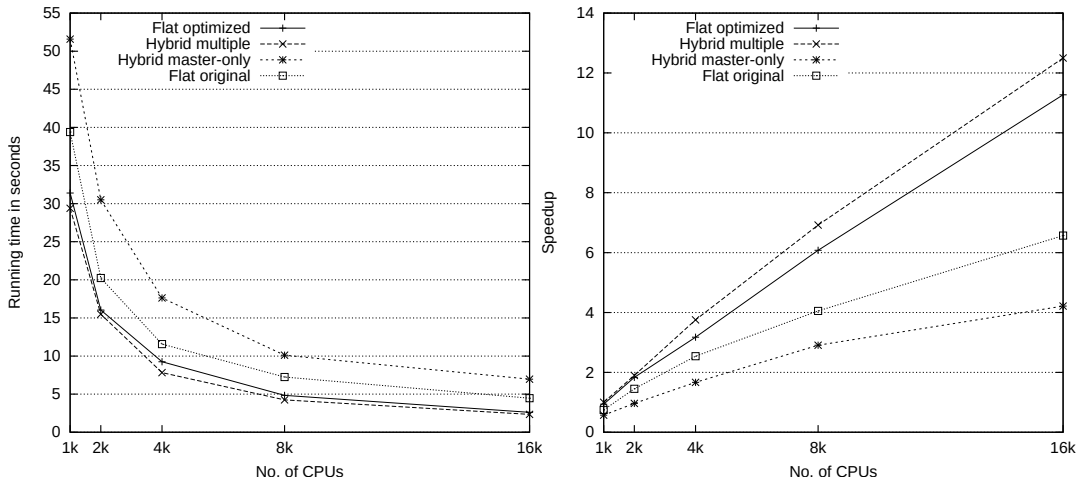


Figure 4.9: A scalability graph starting at 1024 CPU-cores running the stencil operation. In the left graph the running time of every approach is shown and in the right graph every approach is compared to the fastest approach on 1024 CPU-cores namely the **Hybrid multiple**. All jobs consists of 2816 real-space grids all size of 192^3, and the best batch-size has been found for every number of CPU-cores.

At 512 CPU-cores **Hybrid multiple** is faster than **Flat optimized**. The main reason is the difference in the needed communication. **Flat optimized** divides the grids four times more than the **Hybrid multiple**. We did not see this effect in the speedup graph, Figure 4.6, because of the small number of grids. Furthermore, **Hybrid multiple** is better to exploit an increase in grids because of the thread synchronization overhead. The overhead is small and constant, but since the total running time is very small for 32 grids (9 milliseconds with 2048 CPU-cores), the impact of the synchronization overhead is drastically reduced when the number of grids, and thereby the total running time, is increased.

To investigate the scalability of a large job with many real-space grids, we have made a scalability graph beginning at 1k CPU-cores, which allows for a 2816 grid job (Figure 4.9). Again **Hybrid multiple** has the best performance – going from 1k to 16k CPU-cores gives a speedup of approximately 12.5 where 16 would be linear but unobtainable due to the increase in the needed communication. If we compare the running time of **Hybrid multiple** with **Flat original**, we see a 94% performance gain at 16384 CPU-cores.

To further investigate the performance difference between **Hybrid multiple** and **Flat optimized**, we have made a small experiment. We modified **Flat optimized** to statically divide the real-space grids into four sub-groups. It is now possible for all four CPU-cores to work on its own sub-group and the real-space grids will be divided into the same level as in **Hybrid multiple**. The only difference between the two approaches

is that **Flat optimized** uses the virtual mode in Blue Gene/P and **Hybrid multiple** uses threads. It should be noted, however, that in a real GPAW computation this modification does not work, since GPAW requires that every MPI process gets the same subset of every real-space grid, see section 4.3. The experiment is not included in any of the graphs since its performance is identical with the **Hybrid multiple**. Because of the identical performance, we find it reasonable to conclude that the level of real-space partitioning is the sole reason for the performance difference between **Hybrid multiple** and the non-modified **Flat optimized**.

4.7 Summary

In collaboration with the GPAW team at Technical University of Denmark we have managed to improve the performance of a domain specific stencil code when scaling to a very high degree of parallelism. The primary improvements are obtained through the introduction of asynchronous communication which, even in a well balanced system such as the Blue Gene, efficiently improves processor utilization. Furthermore, two hybrid programming approaches have been explored: the hybrid multiple and the master-only approach.

The hybrid programming approach, in which inter-node communication is handled individually by every thread, has shown a positive impact on the performance. By allowing every thread to handle its own inter-node communication, the overhead for thread synchronization remains constant and the application becomes faster than the non-hybrid version.

On the other hand, the alternative hybrid programming approach, in which one thread handles the inter-node communication on behalf of all threads in the process, cannot compete with the non-hybrid version. That is explained by the overhead that is introduced by thread synchronization which grows proportional to the number of grids in the computation.

When comparing our fastest implementation compared to the original implementation, the hybrid programming approach combined with the latency-hiding techniques is 94% faster at 16384 CPU-cores. Translated into utilization this means that CPU utilization grows from 36% to 70%. While latency-hiding is the primary factor for the improvement we observe, the hybrid implementation is still 10% faster than the non-hybrid approach.

We conclude that it is possible to obtain very good performance of the scientific application GPAW but it requires expertise in both distributed and shared memory programming in addition to a lot of work.

Chapter 5

Productivity

High-productivity is the key selling point for a broad range of programming languages and libraries. Some features obviously enhance the productivity – such as interactive execution, interpretation instead of compilation, rich set of standard libraries etc. – but generally, it is difficult to compare the productivity. Nevertheless, we will try by comparing multiple implementations of the stencil computation in GPAW (Chapter 4). For simplicity, we use a 5-point stencil instead of a 9-point stencil and use a fixed number of iterations. The result is a simple application that applies the Jacobi method in a fixed number of iterations. The application is vectorizable, which makes it possible to compare the vector-oriented programming approach with more traditional programming approaches.

Figure 5.1, 5.2, 5.3 and 5.4 is the Jacobi application written in Python/NumPy, MATLAB, ZPL and C, respectively. The Python/NumPy, MATLAB and ZPL implementation uses vectorized operations and express the 5-point stencil with five array subsets that are shifted one element in each direction. As opposed to the C implementation that uses a computation loop with pointer arithmetic explicitly. The similarity of the Python/NumPy and MATLAB version is not a coincidence. Python/NumPy is heavily influenced by MATLAB and strives to provide an open source alternative to MATLAB with similar emphases on high-productivity.

When comparing the mathematical expression of the Jacobi iteration (Section 1.1) with the four implementations it is clear that the Python/NumPy and the MATLAB implementation resemble the mathematical expression the most. Thus, a non-computer expert will find it easier and more productive to use Python/NumPy or the MATLAB. Particularly, the C implementation is not very intuitive for a non-computer expert.

5.1 Parallelization

It is often necessary to introduce parallelization in order to support a large numerical problem because of computation time and data set constrains. The vector-oriented programming model enables the runtime system to handle this parallelization automatically through implicit data parallelism.

```
#Parameters
I     #Number of iterations
A     #Input & Output Matrix
T     #Temporary array
SIZE #Symmetric Matrix Size

#Computation
for i in xrange(I):
  T[:] = (A[1:-1,1:-1] + A[1:-1,:-2] + A[1:-1,2:] + A[:-2,1:-1] \
           + A[2:,1:-1]) / 5.0
  A[1:-1, 1:-1] = T
```

Figure 5.1: Python / NumPy version of Jacobi Iterations.

```
#Parameters
I %Number of iterations
A %Input & Output Matrix
T %Temporary array
SIZE %Symmetric Matrix Size

#Computation
i = 2:SIZE+1;%Center slice vertical
j = 2:SIZE+1;%Center slice horizontal
for n=1:I,
    T(:) = (A(i,j) + A(i+1,j) + A(i-1,j) + A(i,j+1) ...
             + A(i,j-1)) / 5.0;
    A(i,j) = T;
end
```

Figure 5.2: MATLAB version of Jacobi Iterations.

```
1  procedure jacobi();
2  region
3    R    = [1..n,   1..n  ]; -- problem region
4    BigR = [0..n+1, 0..n+1]; --    with borders
5
6  direction
7    north = [-1,  0]; -- cardinal directions
8    east  = [ 0,  1];
9    south = [ 1,  0];
10   west  = [ 0, -1];
11
12 -- Parameters
13 var
14   I -- Number of iterations
15   A -- Input & Output Matrix
16   SIZE -- Symmetric Matrix Size
17   T : [BigR] float -- Temporary array
18 [R] begin
19
20 -- Computation
21   for it := 1 to I do
22     Temp := (A + A@north + A@east + A@south + A@west) / 5.0;
23     A := Temp;
24   end;
25 end;
```

Figure 5.3: ZPL version of Jacobi Iterations.

```
1  //Parameters
2  int I;      //Number of iterations
3  double *A; //Input & Output Matrix
4  double *T; //Temporary array
5  int SIZE;  //Symmetric Matrix Size
6
7  //Computation
8  int gsize = SIZE+2; //Size + borders.
9  for(n=0; n<I; n++)
10 {
11   memcpy(T, A, gsize*gsize*sizeof(double));
12   double *a = A;
13   double *t = T;
14   for(i=0; i<SIZE; ++i)
15   {
16     double *up     = a+1;
17     double *left   = a+gsize;
18     double *right  = a+gsize+2;
19     double *down   = a+1+gsize*2;
20     double *center = t+gsize+1;
21     for(j=0; j<SIZE; ++j)
22       *center++ = (*center + *up++ + *left++ + *right++ + *down++) \
23                   / 5.0;
24     a += gsize;
25     t += gsize;
26    }
27    memcpy(A, T, gsize*gsize*sizeof(double));
28 }
```

Figure 5.4: Sequential version of the Jacobi Iterations in C.

ZPL[29] facilitates implicit data parallelism natively without the need to rewrite any code. In C, on the other hand, the programmer has to express parallelization explicitly when programming. There exist libraries and languages extensions that try to make parallelization easier though. In this chapter, we will explore two industrial standards for parallel programming in C: one for shared memory, OpenMP, and one for distributed shared memory, MPI. Common for both approaches is that the programmer needs to implement a parallelization strategy for the domain decomposition, local / global variable access, communication etc.

MATLAB do not facilitate implicit data parallelism natively. However, some extensions exist that seamlessly introduce data parallelism[97, 31, 32, 57]. In Chapter 7, we will introduce implicit data parallelism to the Python/NumPy programming language.

5.1.1 OpenMP

Often shared memory programming uses some kind of threading in order to utilize multi CPU-cores. OpenMP is very popular for doing shared memory programming in C. Figure 2.2 from Section 2.1.1 shows a C implementation of the Jacobi application that uses OpenMP to parallelize the computation loop. The code is almost identical with the sequential version (Figure 5.4) thus, in this case, OpenMP does not introduce much complexity because the strategy for dividing the workload between the threads is straightforward. However, it was necessary to make the loop iterations non-dependent.

5.1.2 MPI

Distributed memory programming is typically more complex than shared memory programming because distributed memory programming requires differentiation between local and remote memory. Message passing is a technique for efficiently coordinating and communicating local data structures and MPI is the industrial standard for SPMD parallel programming. Figure 5.5 shows a version of the Jacobi application that uses MPI to parallelize the computation loop. Even though the implementation uses data parallelism with a very simple domain decomposition, the implementation is a lot more complex than the sequential and the OpenMP version.

5.1.3 MPI and OpenMP

In order to introduce even further parallelism we could use hybrid programming. Additionally, communication latency-hiding is essential to optimize the Jacobi application implementation fully.

Figure 5.6 shows an optimized version of the Jacobi application that utilizes both hybrid programming and communication latency-hiding. In order to overlap communucation with computation, it uses the double buffering technique. It starts by initiating the communication using non-blocking MPI functions; then it computes all elements that do not depend on remotely located data. Finally, it waits for the communication to finish and compute the rest of the elements. The code is clearly more complex than

```
1  //Parameters
2  int I;      //Number of iterations
3  double *A; //Input & Output Matrix (local)
4  double *T; //Temporary array (local)
5  int SIZE;  //Symmetric Matrix Size (local)
6
7  //Computation
8  int gsize = SIZE+2; //Size + borders.
9  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
10 MPI_Comm_size(MPI_COMM_WORLD, &worldsize);
11 MPI_Comm comm;
12 int periods[] = {0};
13 MPI_Cart_create(MPI_COMM_WORLD, 1, &worldsize,
14                 periods, 1, &comm);
15 int l_size = SIZE / worldsize;
16 if(myrank == worldsize-1)
17    l_size += SIZE % worldsize;
18 int l_gsize = l_size + 2;//Size + borders.
19 for(n=0; n<I; n++)
20 {
21   int p_src, p_dest;
22   //Send/receive - neighbor above
23   MPI_Cart_shift(comm,0,1,&p_src,&p_dest);
24   MPI_Sendrecv(A+gsize,gsize,MPI_DOUBLE,
25                p_dest,1,A,gsize, MPI_DOUBLE,
26                p_src,1,comm,MPI_STATUS_IGNORE);
27   //Send/receive - neighbor below
28   MPI_Cart_shift(comm,0,-1,&p_src,&p_dest);
29   MPI_Sendrecv(A+(l_gsize-2)*gsize,
30                gsize,MPI_DOUBLE,
31                p_dest,1,A+(l_gsize-1)*gsize,
32                gsize,MPI_DOUBLE,
33                p_src,1,comm,MPI_STATUS_IGNORE);
34   memcpy(T, A, l_gsize*gsize*sizeof(double));
35   double *a = A;
36   double *t = T;
37   for(i=0; i<SIZE; ++i)
38   {
39     int a = i * gsize;
40     double *up     = &A[a+1];
41     double *left   = &A[a+gsize];
42     double *right  = &A[a+gsize+2];
43     double *down   = &A[a+1+gsize*2];
44     double *center = &T[a+gsize+1];
45     for(j=0; j<SIZE; ++j)
46       *center++ = (*center + *up++ + *left++ + *right++ + *down++) \
47                   / 5.0;
48   }
49   MPI_Barrier(MPI_COMM_WORLD);
50 }
```

Figure 5.5: Parallel version of the Jacobi Iterations in C using MPI.

the sequential version. Compared to Python/NumPy the C code has more than six times the number of line codes. It uses explicit message parsing where the programmer must implement the domain decomposition and parallelization strategy. Memory access through pointer arithmetic and synchronization between threads and processes is handled completely by the programmer.

5.2 Summary

Overall, we conclude that the programming productivity with the vector-oriented programming model is superior for some applications. Particularly, when comparing explicit versus implicit parallelism. The vector-oriented programming model provides full knowledge of data distribution and parallelization to all participating processors, which makes it possible for the runtime system to execute vector operations seamlessly in parallel without further assistance from the user. Additionally, the processors need not communicate when performing data dependency analysis and scheduling optimizations at runtime.

However, the model also reduces the programmability because the user is restricted to vector operations, which makes it inconvenient to perform task parallelism and conditional computations. It is therefore essential that the application is vectorizable in the sense that it is possible to express the application using high-level vector operations primarily. This is the main reason why we insist on focusing on scientific application rather than applications in general. Typically, a scientific application is easy to vectorize, which is evident when looking at the popularity of programming languages such as MATLAB and Python/NumPy. Furthermore, the scientific community has a long tradition of utilizing high-level array operations through libraries such as BLAS, LAPACK and FFTW.

```
1  //Parameters
2  int I;      //Number of iterations
3  double *A; //Input & Output Matrix (local)
4  double *T; //Temporary array (local)
5  int SIZE;  //Symmetric Matrix Size (local)
6
7  //Computation
8  int gsize = SIZE+2; //Size + borders.
9  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
10 MPI_Comm_size(MPI_COMM_WORLD, &worldsize);
11 MPI_Comm comm;
12 int periods[] = {0};
13 MPI_Cart_create(MPI_COMM_WORLD, 1, &worldsize,
14                 periods, 1, &comm);
15 int l_size = SIZE / worldsize;
16 if(myrank == worldsize-1)
17    l_size += SIZE % worldsize;
18 int l_gsize = l_size + 2;//Size + borders.
19 for(n=0; n<I; n++)
20 {
21   int p_src, p_dest;
22   MPI_Request reqs[4];
23
24   //Initiate send/receive - neighbor above
25   MPI_Cart_shift(comm, 0, 1, &p_src, &p_dest);
26   MPI_Isend(A+gsize, gsize, MPI_DOUBLE, p_dest,
27             1, comm, &reqs[0]);
28   MPI_Irecv(A, gsize, MPI_DOUBLE, p_src,
29             1, comm, &reqs[1]);
30
31   //Initiate send/receive - neighbor below
32   MPI_Cart_shift(comm, 0, -1, &p_src, &p_dest);
33   MPI_Isend(A+(l_gsize-2)*gsize, gsize,
34             MPI_DOUBLE,
35             p_dest, 1, comm, &reqs[2]);
36   MPI_Irecv(A+(l_gsize-1)*gsize, gsize,
37             MPI_DOUBLE,
38             p_src, 1, comm, &reqs[3]);
39
40   //Handle the non-border elements.
41   memcpy(T+gsize, A+gsize, l_size*gsize*sizeof(double));
42   #pragma omp parallel for shared(A,T)
43   for(i=1; i<l_size-1; ++i)
44     compute_row(i,A,T,SIZE,gsize);
45
46   //Handle the upper and lower ghost line
47   MPI_Waitall(4, reqs, MPI_STATUSES_IGNORE);
48   compute_row(0,A,T,SIZE,gsize);
49   compute_row(l_size-1,A,T,SIZE,gsize);
50
51   memcpy(A+gsize, T+gsize, l_size*gsize*sizeof(double));
52 }
53 MPI_Barrier(MPI_COMM_WORLD);
```

Figure 5.6: Parallel version in C using MPI and OpenMP. In this code we use the function `compute_row()` to compute one row. The implementation of `compute_row()` is simuliar to line 15 to 26 in Figure 5.4.

Chapter 6

Numerical Python

Python is a popular programming language in the Computational Science and Engineering community. It prioritizes high-productivity over high-performance, which is one of the reasons for its popularity. However, it also means that the performance of applications purely written in Python is inadequate for must scientific purposes. In order to address this performance issue the scientific community uses Python as the gluing language of external libraries written in high-performance languages such as C and FORTRAN. An example is SciPy[61], which is a very popular Python library that provides a massive collection of optimized operations implemented in C, C++ and FORTRAN. The operations are available in Python through a multidimensional array object that nicely integrates with the Python language. The array object originates from the Python library Numerical Python (NumPy)[82], which is a sub-project in SciPy and the basis for most scientific applications in Python.

6.1 Universal Functions

Beside providing a broad range of high-level array operations, such as LU factorization, FFT, and Matrix Multiplication, NumPy provides generic element-wise vector operations. By using these vector operations, NumPy takes advantage of the performance of C while maintaining the high abstraction level of Python. In NumPy these element-wise vector operations are called universal functions. A universal function (ufunc) is an element-wise vector operation that computes all elements in an array-view independently. Applying an ufunc operation on a whole array is semantically equivalent to performing the ufunc operation on each array element individually. Using ufunc can result in a significant performance boost compared to native Python because the computation-loop is executed in C.

6.1.1 Function broadcasting

To make ufunc more flexible it is possible to use arrays with different number of dimensions. To utilize this feature the size of the dimensions must either be identical or

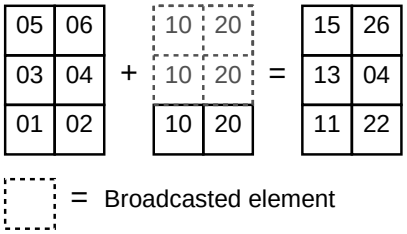


Figure 6.1: Universal function broadcasting. The ufunc `addition` is applied on a 3x2 array and a 1x2 array. The first dimension of the 1x2 array is broadcasted to the size of the first dimension of the 3x2 array. The result is a 3x2 array in which the two arrays are added together in an element-by-element fashion.

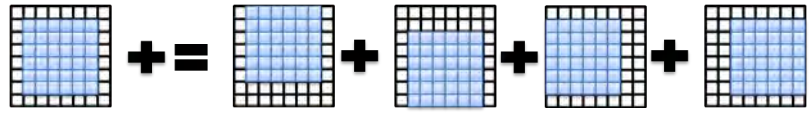


Figure 6.2: Python expression of a simple 5-point stencil computation example.

have the length one. When the ufunc is applied, all dimensions with a size of one will be *broadcasted* in the NumPy terminology. That is, the array will be duplicated along the *broadcasted* dimension (Figure 6.1). It is possible to implement many array operations efficiently in Python by combining NumPy's ufunc with more traditional numerical functions like matrix multiplication, factorization etc.

6.2 Array Syntax and Views

NumPy uses an array syntax that is based on the Python list syntax. The arrays are indexed positionally, 0 through length – 1, where negative indexes is used for indexing from the end of the array instead of the beginning. Like the list syntax in Python, it is possible to index multiple elements. All indexing that represents more than one element returns a view of the elements rather than a new copy of the elements. It is this view semantic that makes it possible to implement a stencil operation as demonstrated in Figure 6.2. In order to force a real array copy rather than a new array reference NumPy provides the `copy` method.

6.3 Interfaces

The primary interface in NumPy is a Python interface and it is possible to use NumPy exclusively from Python. NumPy also provides a C interface in which it is possible to access the same functionality as in the Python interface. Additionally, the C interface also allows programmers to access low level data structures like pointers to array data

and thereby provides the possibility to implement arbitrary array operations efficiently in C. The two interfaces may be used interchangeably through the Python program.

Chapter 7

Distributed Numerical Python

After my work with GPAW I explored the possibility of seamlessly utilize distributed memory architectures directly in Python. GPAW make use of the Python module NumPy. However, since NumPy do not support distributed memory parallel programming, GPAW use of NumPy is limited to local operations inside MPI-processes. It is in this context the idea of a parallel version of NumPy emerged.

In this section, I will present the development of a new version of NumPy that target scalable architectures. We call this project Distributed Numerical Python (DistNumPy) and it is a key contribution of this thesis. I divide the development of DistNumPy into four stages and describe each development stage separately.

Stage One The basic implementation of DistNumPy that has very limited array-view support.

Stage Two Introduction of full array-view support.

Stage Three Introduction of communication latency hiding.

Stage Four Introduction of PGAS-style programming.

However, before describing all four development stages I will give an overall introduction of DistNumPy.

7.1 Introduction

Distributed Numerical Python (DistNumPy) is a library for doing numerical computation in Python that targets scalable distributed memory architectures. Replacing NumPy with DistNumPy enables the user to write sequential Python programs that seamlessly utilize distributed memory architectures. This feature is obtained by introducing a new backend for NumPy arrays that distribute data amongst the nodes in a distributed memory multi-processor. All operations on this new array will seek to utilize all available processors. The array itself is distributed between multiple processors in order to support larger arrays than a single node can hold in memory.

The only difference in the API of NumPy and DistNumPy is the array creation routines. DistNumPy allow both distributed and non-distributed arrays to co-exist thus the user must specify, as an optional parameter, if the array should be distributed. The following illustrates the only difference between the creation of a standard array and a distributed array:

```
A = numpy.array([1,2,3])#Non-Distributed
B = numpy.array([1,2,3], dist=True)#Distributed
```

7.1.1 Target architectures

NumPy supports a long range of architectures from the widespread x86 to the specialized Blue Gene architecture. However, NumPy is incapable of utilizing distributed memory architectures like Blue Gene supercomputers or clusters of x86 machines. The target of DistNumPy is to close this gap and fully support and utilize distributed memory architectures.

7.1.2 Motivated by Related Work

Libraries and programming languages that support parallelism in a high productive manner is a well-known concept. The existing tools either seek to provide optimal performance in parallel applications or, like DistNumPy, seek to ease the task of writing parallel applications. In a perfect framework, all parallelism introduced by the framework is completely transparent to the user while the performance and scalability achieved is optimal. However, most frameworks require the user to specify some kind of parallelism – either explicitly by using parallel directives or implicitly by using parallel data structures. Libraries and programming languages that support parallelization on distributed memory architectures is a well-known concept. The existing tools either seek to provide optimal performance in parallel applications or, like DistNumPy, seek to ease the task of writing parallel applications.

The library ScaLAPACK[**?**, 15], requires knowledge in distributed shared memory programming and it is the responsibility of the programmer to ensure that the global data structures, e.g. matrices and vectors, comply with the distribution layout specified by ScaLAPACK.

Another library, Global Arrays[79], introduces a global array, which makes the data distribution transparent to the user. It also supports efficient parallel operations and provides a higher level of abstraction than ScaLAPACK. However, the programmer must still explicitly coordinate the multiple processes that are involved in the computation. The programmer must specify which region of a global array is relevant for a given process.

Both ScaLAPACK and Global Arrays may be used from within Python and can even be used in combination with NumPy, but it is only possible to use NumPy locally and not with distributed operations. A more closely integrated Python project IPython supports parallelized NumPy operations. IPython introduces a distributed NumPy array much like the distributed array that is introduced in this work. Still, the user-application

must use the MPI framework and the user has to differentiate between the running MPI-processes.

A higher level of abstraction is found in projects where the execution, seen from the perspective of the user, is represented as a sequential algorithm. In effect, such project provides a framework where the two most notorious types of parallel programming bugs, data races and deadlocks, simply do not exist. The High Performance Fortran programming language provides such an abstraction level. High Performance Fortran introduces parallelism primarily with vector operations, which, in order to archive good performance, must be aligned by the user to reduce communication. A lot of work has been put into eliminating this alignment issue either at compile-time or run-time [67] [20] [11] but the popularity of High Performance Fortran is still very limited.

The Simple Parallel R INTerface (SPRINT)[55] is a parallel framework for the programming language R. The abstraction level in SPRINT is similar to DistNumPy in the sense that the distribution and parallelization is completely transparent to the user.

7.2 The Basic Implementation

DistNumPy is a new version of NumPy that parallelizes array operations in a manner completely transparent to the user - from the perspective of the user, the difference between NumPy and DistNumPy is minimal. DistNumPy can use multiple processors through the communication library Message Passing Interface (MPI)[50]. However, we have chosen not to follow the standard MPI approach where the same user-program is executed on all MPI-processes. This is because the standard MPI approach requires the user to differentiate between the MPI-processes, e.g. sequential areas in the user-program must be guarded with a branch based on the MPI-rank of the process. In DistNumPy MPI communication must be fully transparent and the user needs no knowledge of MPI or any parallel programming model. However, the user is required to use the array operations in DistNumPy to obtain any kind of speedup. We think this is a reasonable requirement since it is also required by NumPy.

7.2.1 Interfaces

There are two programming interfaces in NumPy - one in Python and one in C. We aim to support the complete Python interface and a great subset of the C interface. However, the part of the C interface that involves direct access to low level data structures will not be supported. It is not feasible to return a C-pointer that represents the elements in a distributed array.

7.2.2 Data layout

Two-Dimensional Block Cyclic Distribution is a very popular distribution scheme and it is used in numerical libraries like ScaLAPACK and LINPACK[39]. It supports matrices and vectors and has a good load balance in numerical problems that have a diagonal computation workflow e.g. Gaussian elimination. The distribution scheme works by

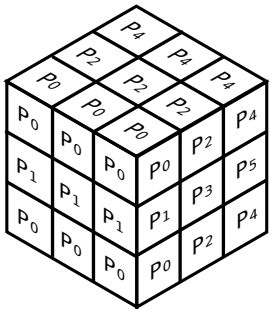


Figure 7.1: The N-Dimensional Block Cyclic Distribution of a matrix on a 3 x 2 x 1 grid of processors.

arranging all MPI-processes in a two dimensional grid and then distributing data-blocks in a round-robin fashion either along one or both grid dimensions (Figure 2.10).

NumPy is not limited to matrices and vectors as it supports arrays with an arbitrary number of dimensions[1]. DistNumPy therefore use a more generalized N-Dimensional Block Cyclic Distribution inspired by High Performance Fortran[73], which supports an arbitrary number of dimensions (Figure 7.1). Instead of using a fixed process grid, we have a process grid for every number of dimensions. This works well when operating on arrays with the same number of dimensions but causes problems otherwise. For instance in a matrix-vector multiplication the two arrays are distributed on different process grid and may therefore require more communication. ScaLAPACK solves the problem by distributing vectors on two-dimensional process grids instead of one-dimensional process grids, but this will result in vector operations that cannot utilize all available processors. An alternative solution is to redistribute the data between a series of identically leveled BLAS operations using a fast runtime redistribution algorithm like [87] demonstrates.

7.2.3 Operation dispatching

The MPI-process hierarchy in DistNumPy has one MPI-process (master) placed above the others (slaves). All MPI-processes run the Python interpreter but only the master executes the user-program, the slaves will block at the `import numpy` statement.

The following describes the flow of the dispatching:

1. The master is the dispatcher and will, when the user applies a numpy command on a distributed array, compose a message with meta-data describing the command.

2. The message is then broadcasted from the master to the slaves with a blocking MPI-broadcast. It is important to note that the message only contains meta-data and not any actual array data.

[1]The number of dimensions NumPy supports is defined at compile time. The default value is 16 dimensions.

3. After the broadcast, all MPI-processes will apply the command on the sub-array they own and exchange array elements as required (Point-to-Point communication).

4. When the command is completed, the slaves will wait for the next command from the master and the master will return to the user's python program. The master will return even though some slaves may still be working on the command, synchronization is therefore required before the next command broadcast.

7.2.4 Views

In NumPy an array does not necessarily represent a complete contiguous block of memory. An array is allowed to represent a subpart of another array i.e. it is possible to have a hierarchy of arrays where only one array represent a complete contiguous block of memory and the other arrays represent a subpart of that memory.

Inspired by NumPy, DistNumPy implements an array hierarchy where distributed arrays are represented by the following two data structures.

- **Array-base** is the base of an array and has direct access to the content of the array in main memory. An array-base is created with all related meta-data when the user allocates a new distributed array, but the user will never access the array directly through the array-base. The array-base always describes the whole array and its meta-data such as array size and data type are constant.

- **Array-view** is a view of an array-base. The view can represent the whole array-base or only a sub-part of the array-base. An array-view can even represent a non-contiguous sub-part of the array-base. An array-view contains its own meta-data that describe which part of the array-base is visible and it can add non-existing 1-length dimensions to the array-base. The array-view is manipulated directly by the user and from the users perspective the array-view is the array.

Array-views are not allowed to refer to each other, which means that the hierarchy is flat with only two levels: array-base below array-view. However, multiple array-views are allowed to refer to the same array-base. This hierarchy is illustrated in Figure 7.2.

7.2.5 Non-Aligned Array Operations

Array views gives rise to a number of important performance challenges when combined with data parallelism where the shared data is distributed across multiple processes. The problem is that operations on views may translate into *non-aligned* distributed array operations, which are difficult to handle efficiently. We define an *aligned* distributed array operation as an operation on arrays that are distributed in a conformable manner, i.e. the arrays use identical data distribution. A non-aligned distributed array operation is then an operation without this property. For now, we will simply ignore this problem – in the basic implementation we only supports *aligned* distributed array operations. In the next implementation stage of DistNumPy, we introduce full array-view support (see section 7.3 for further discussion of the performance implications).

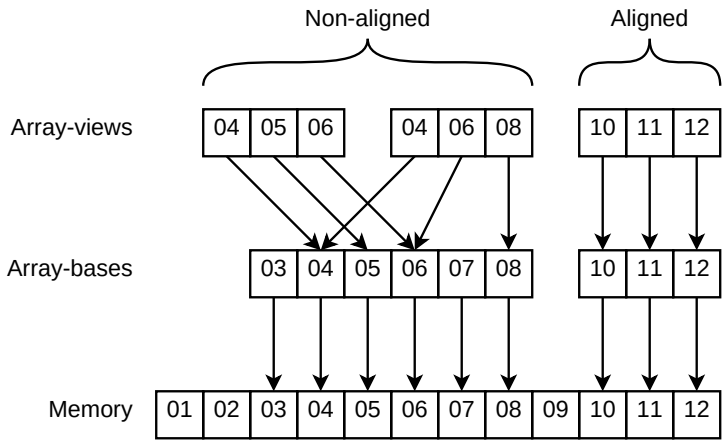


Figure 7.2: Reference hierarchy between the two array data structures and the main memory. Only the three array-views at top of the hierarchy are visible from the perspective of the user.

7.2.6 Parallel BLAS

NumPy supports BLAS operations on vectors and matrices. DistNumPy therefore implements a parallel version of BLAS inspired by PBLAS from the ScaLAPACK library. Since DistNumPy uses the same data-layout as ScaLAPACK, it would be straightforward to use PBLAS for all parallel BLAS operations. However, to simplify the installation and maintenance of DistNumPy we have chosen to implement our own parallel version of BLAS. We use SUMMA[47] for matrix multiplication, which enable us to use the already available BLAS library locally on the MPI-processes. SUMMA is only applicable on complete array-views and we therefore use a straightforward implementation that computes one element at a time if partial array-views are involved in the computation.

7.2.7 Universal function

In DistNumPy, the implementation of ufunc uses three different scenarios.

1. In the simplest scenario we have an aligned match between all elements in the array-views and applying an ufunc does not require any communication between MPI-processes. The scenario is applicable when the ufunc is applied on complete array-views with identical shapes.

2. In the second scenario the array-views must represent a continuous part of the underlying array-base. The computation is parallelized by the data distribution of the output array and data blocks from the input arrays are fetched when needed. We use non-blocking one-side communication (`MPI_Get`) when fetching data blocks, which makes it possible to compute one block while fetching the next block (Figure 4.4).

```
1 from numpy import *
2 (x, y) = (empty([S], dist=True), \
3           empty([S], dist=True))
4 (x, y) = (random(x), random(y))
5 (x, y) = (square(x), square(y))
6 z = (x + y) < 1
7 print add.reduce(z) * 4.0 / S #The result
```

Figure 7.3: Computing Pi using Monte Carlo simulation. `S` is the number of samples used. We have defined a new ufunc (`ufunc_random`) to make sure that we use an identical random number generator in all benchmarks. The ufunc uses "`rand()/(double)RAND_MAX`" from the ANSI C standard library (`stdlib.h`) to generate numbers.

3. The final scenario does not use any simplifications and works with any kind of array-view. It also uses non-blocking one-side communication but only one element at a time.

7.2.8 Examples

To evaluate DistNumPy we have implemented three Python programs that all make use of NumPy's vector-operations (ufunc). They are all optimized for a sequential execution on a single CPU and the only program change we make, when going from the original NumPy to our DistNumPy, is the array creation argument `dist`. A walkthrough of a Monte Carlo simulation is presented as an example of how DistNumPy handles Python executions.

Monte Carlo simulation

We have implemented a trivial Monte Carlo Pi simulation using NumPy's ufunc. The implementation is a translation of the Monte Carlo simulation included in the benchmark suite SciMark 2.0[86], which is written in Java. It is very simple and uses two vectors with length equal the number of samples used in the calculation. Because of the memory requirements, this drastically reduces the maximum number of samples. Combining multiple simulations will allow more samples but we will only use one simulation. The implementation is included in its full length (Figure 7.3) and the following is a walkthrough of a simulation (the bullet-numbers represents line numbers):

1: All MPI-processes interpret the `import` statement and initiate DistNumPy. Besides calling `MPI_Init()` the initialization is identical to the original NumPy but instead of returning from the import statement, the slaves, MPI-processes with rank greater than zero, listen for a command message from the master, the MPI-process with rank zero.

```
1 h = zeros(shape(B), float, dist=True)
2 dmax = 1.0
3 AD = A.diagonal()
4 while(dmax > tol):
5   hnew = h + (B - (A * h).sum()) / AD
6   tmp = absolute((h - hnew) / h)
7   dmax = maximum.reduce(tmp)
8   h = hnew
9 print h #The result
```

Figure 7.4: Iteratively Jacobi solver for matrix `A` with solution vector `B` both are distributed arrays. The `import` statement and the creation of `A` and `B` is not included here. `tol` is the maximum tolerated value of the diagonal-element with the highest value (`dmax`).

2-3: The master sends two `CREATE_ARRAY` messages to all slaves. The two messages contain an array shape and unique identifier (UID), which in this case identifies `x` and `y`, respectively. All MPI-processes allocate memory for the arrays and stores the array information.

4: The master sends two `UFUNC` messages to all slaves. Each message contains a UID and a function name `ufunc_random`. All MPI-processes apply the function on the array with the specified UID. A pointer to the function is found by calling `PyObject_Get AttrString` with the function name. It is thereby possible to support all ufuncs from NumPy.

5: Again the master sends two `UFUNC` messages to all slaves but this time with function name `square`.

6: The master sends a `UFUNC` messages with function name `add` followed by a `UFUNC` messages with function name `less_than`. The scalar `1` is also in the message.

7: The master sends a `UFUNC_REDUCE` messages with function name `add`. The result is a scalar, which is not distributed, and the master therefore solely computes the remainder of the computation and print the result. When the master is done a `SHUTDOWN` message is sent to the slaves and the slaves call `exit(0)`.

Jacobi method

The Jacobi method is an algorithm for determining the solutions of a system of linear equations. It is an iterative method that uses a spitting scheme to approximate the result.

Our implementation uses ufunc operations in a while-loop until it converges. Most of the implementation is included here(Figure 7.4).

Table 7.1: Hardware specifications

CPU	Intel Core 2 Quad Q9400	Intel Nehalem E5520
CPU Frequency	2.26 GHz	2.66 GHz
CPU per node	1	2
Cores per CPU	4	4
Memory per node	8 GB @ 6.5 GB/s	24 GB @ 25.6 GB/s
Number of nodes	8	8
Network	Gigabit Ethernet	Gigabit Ethernet

Newtonian N-body simulation

A Newtonian N-body simulation is one that studies how bodies, represented by a mass, a location, and a velocity, move in space according to the laws of Newtonian physics. We use a straightforward algorithm computing all body-body interactions. The NumPy implementation is a direct translation of a MATLAB program[26].

7.2.9 Experiments

In this section, we will conduct performance benchmarks on DistNumPy and NumPy[2]. We will benchmark the three Python programs presented in Section 7.2.8. All benchmarks are executed on two different Linux clusters. (Table 7.1).

Our experiments consist of a speedup benchmark, which we define as an execution time comparison between a sequential execution with NumPy and a parallelized execution with DistNumPy while the input is identical.

Monte Carlo simulation

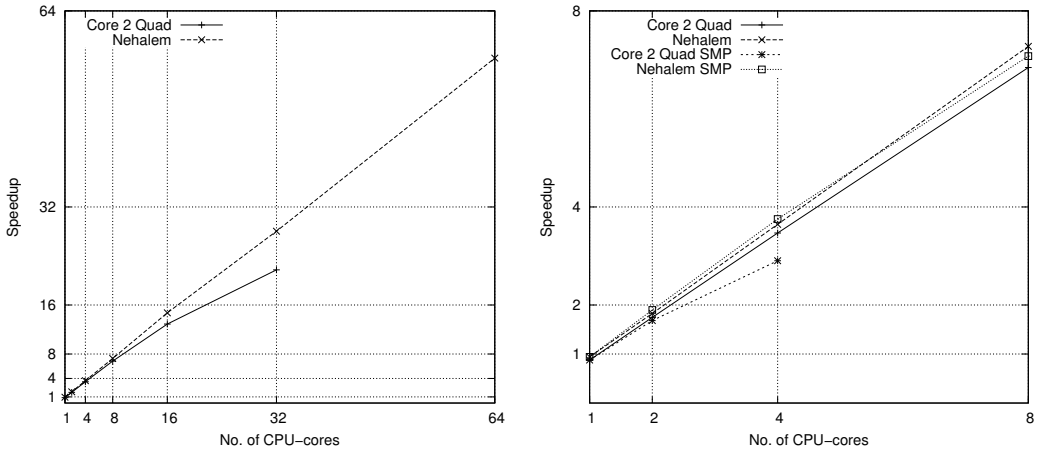
A Distributed Monte Carlo simulation is embarrassingly parallel and requires a minimum of communication. This is also the case when using DistNumPy because ufuncs are only applied on identically shaped arrays and it is therefore the simplest ufunc scenario. Additionally, the implementation is CPU-intensive because a complex ufunc is used as random number generator.

The result of the speedup benchmark is illustrated in Figure 7.5. We see a close to linear speedup for the Nehalem cluster - a CPU utilization of 88% is achieved on 64 CPU-cores. The penalty of using multiple CPU-cores per node is noticeable on the Core 2 architecture - a CPU utilization of 68% is achieved on 32 CPU-cores.

Jacobi method

The dominating part of the Jacobi method, performance-wise, is the element-by-element multiplication of `A` and `h` (Figure 7.4 line 5). It consists of $O(n^2)$ operations where as all the other operations only consist $O(n)$ operations. Since scalar-multiplication is a very simple operation, the dominating ufunc in the implementation is memory-intensive.

[2]NumPy version 1.3.0



(a) (b)

Figure 7.5: Speedup of the Monte Carlo simulation. In graph (a) the two architectures uses a minimum number of CPU-cores per node. Added in graph (b) is the result of using multiple CPU-cores on a single node (SMP).

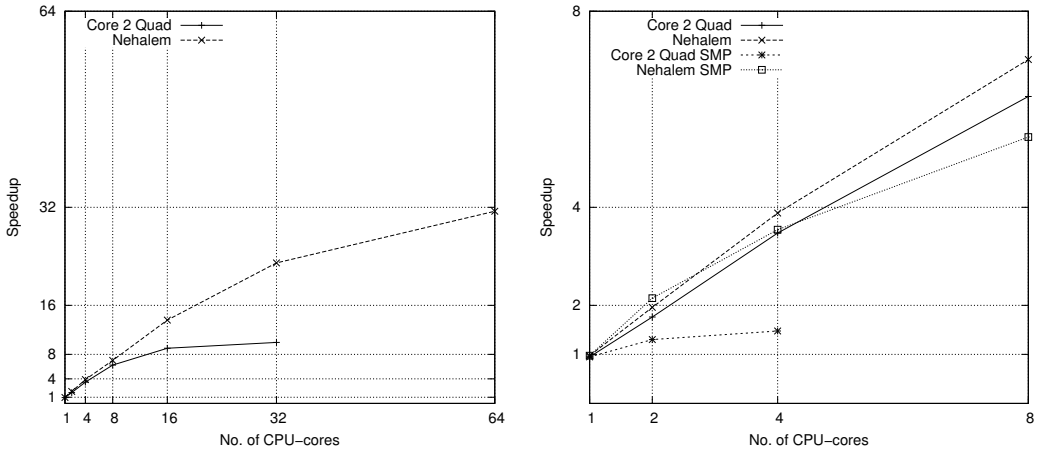
The result of the speedup benchmark is illustrated in Figure 7.6. We see a good speedup with 8 CPU-cores and to some degree also with 16 Nehalem CPU-cores. However, the CPU utilization when using more than 16 CPU-cores is very poor. The problem is memory bandwidth - since we use multiple CPU-cores per node when using more than 8 CPU-cores, the aggregated memory bandwidth of the Core 2 cluster does only increase up to 8 CPU-cores. The Nehalem cluster is a bit better because it has two memory buses per node, but using more than 16 CPU-cores will not increase the aggregated memory bandwidth.

Newtonian N-body simulation

The result of the speedup benchmark is illustrated in Figure 7.7. Compared to the Jacobi method we see a similar speedup and CPU utilization. This is expected because the dominating operations are also simple ufuncs. Even though there are some matrix-multiplications, which have a great scalability, it is not enough to significantly boost the overall scalability.

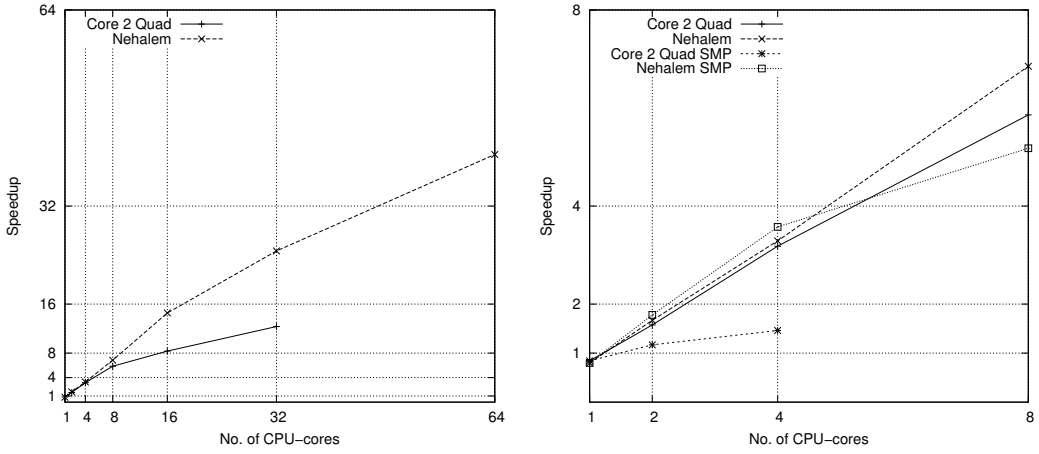
Alternative programming language

DistNumPy introduces a performance overhead compared to a lower-level programming language such as C/C++ or Fortran. To investigate this overhead we have implemented the Jacobi benchmark in C. The implementation uses the same sequential algorithm as the NumPy and DistNumPy implementations.



(a) (b)

Figure 7.6: Speedup of the Jacobi solver. In graph (a) the two architectures uses a minimum number of CPU-cores per node. Added in graph (b) is the result of using multiple CPU-cores on a single node (SMP).



(a) (b)

Figure 7.7: Speedup of Newtonian N-body simulation. In graph (a) the two architectures uses a minimum number of CPU-cores per node. Added in graph (b) is the result of using multiple CPU-cores on a single node (SMP).

Executions on the both architectures show that DistNumPy and NumPy is roughly 50% slower than the C implementation when executing the Jacobi method on one CPU-core. This is in rough runtime numbers: 21 seconds for C, 31 seconds for NumPy and 32 seconds for DistNumPy.

7.2.10 Conclusion

The benchmarks clearly show that DistNumPy has both good performance and scalability when execution is not bound by the memory bandwidth, which is evident from looking at the CPU utilization when only one CPU-core per node is used. As expected the scalability of the Monte Carlo simulation is better than the Jacobi and the N-body computation because of the reduced communication requirements and more CPU-intensive ufunc operation.

The scalability of the Jacobi and the N-body computation is drastically reduced when using multiple CPU-cores per node. The problem is the complexity of the ufunc operations. As opposed to the Monte Carlo simulation, which makes use of a complex ufunc, the Jacobi and the N-body computation only use simple ufuncs e.g. add and multiplication.

As expected the performance of the C implementation is better than the DistNumPy implementation. However, by utilizing two CPU-cores it is possible to outperform the C implementation in the case of the Jacobi method. This is not a possibility in the case of the Monte Carlo simulation where the algorithm does not favor vectorization.

The basic implementation of DistNumPy demonstrates that it is possible to implement a parallelized version of NumPy that seamlessly utilize distributed memory architectures. A CPU utilization of 88% is achieved on a 64 CPU-core Nehalem cluster running a CPU-intensive Monte Carlo simulation. A more memory-intensive N-body simulation achieves a CPU utilization of 91% on 16 CPU-cores but only 63% on 64 CPU-cores. Similar a Jacobi solver achieves a CPU utilization of 85% on 16 CPU-cores and 50% on 64 CPU-cores.

7.3 Full Array View Support

The second development stage of DistNumPy introduces a model for managing abstract data structures that map to arbitrary distributed memory architectures. We use this model in order to implement full array view support in DistNumPy though the model is not restricted to DistNumPy.

7.3.1 Introduction

It is difficult to achieve scalable performance in data-parallel applications where the programmer manipulates abstract data structures rather than directly manipulating memory. On distributed memory architectures, such abstract data-parallel operations may require communication between nodes. Therefore, the underlying system has to handle communication efficiently without any help from the user. Our data model splits

data blocks into two sets – local data and remote data – and schedules the sub-block by availability at runtime.

High Performance Fortran (HPF)[73] and ZPL[29] are two well-known examples of data-parallel programming languages that supports abstract data structures. HPF is a Fortran-based data-parallel programming language that requires static compilation for distributed-memory systems[63]. To obtain good parallel performance the user must *align* arrays together to reduce communication[11]. Our data model manages computation and communication of abstract data structures at runtime, which enables on-the-fly data dependency analysis. Using our model the user will not have to *align* arrays in order to obtain good parallel performance.

Data-Parallel Applications

Data-parallel applications are a class of applications that make use of data parallelism – either explicitly handled by the programmer or implicitly handled by the programming language or library. In this work, we focus on data-parallel applications written in a high-productivity language where the programming language, scientific library, and/or runtime system handles the data parallelism seamlessly. We target applications with the following properties:

- The application uses high-level array operations instead of explicitly programmed *for* loops.
- The application uses data parallelism to execute vector/array operation in parallel.
- In order to utilize distributed memory architectures, the application distributes data evenly across process using a static distribution scheme.
- The application uses data structures that maps to arbitrary distributed memory, e.g. by using data structures, such as array views, that may refer to parts of the same underlying data.

It is no coincidence that DistNumPy has all four properties. The main motivation for developing the model was clearly full view support in DistNumPy but the model is general enough to support other parallel frameworks.

7.3.2 Managing Non-Aligned Array Operations

Managing overlapping data structures, aka array-view, for data-parallel applications on distributed memory architectures gives rise to a number of important performance challenges. For example, a 3-point stencil application uses three array-views, A, B and C, to express a stencil (Figure 7.8). When executing on two processes the two underlying array-bases, M and N, are distributed according to Figure 7.9. It is clear that A and C does not map directly to the underlying array-bases M and N. Thus, the result is a non-aligned array operation. In order to execute such an application the two processes must exchange data blocks, which mean commutation when executing on

```
1 M = numpy.array([1,2,3,4,5,6],dist=True)
2 N = numpy.empty((6),dist=True)
3 A = M[2:]
4 B = M[0:4]
5 C = N[1:5]
6 C = A + B
```

Figure 7.8: This is an example of a small 3-point stencil application.

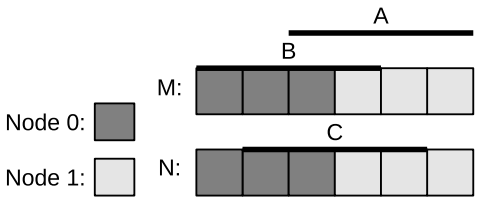


Figure 7.9: The data layout of the two arrays M and N and the three array-views A, B and C in the 3-point stencil application (Figure 7.8). The arrays are distributed between two nodes using a block-size of three.

a distributed memory architecture. Therefore, an efficient data structure model that minimizes communication is vital for the parallel performance.

The main contribution in this work is a model for managing non-aligned array operations efficiently. We introduce a hierarchy of data structures that makes it possible to divided non-aligned array operations into aligned blocks at runtime while minimizing the total amount of communication.

The model consists of three kinds of data blocks: base-blocks, view-blocks and sub-view-blocks, which make up a three level abstraction hierarchy (Figure 7.10).

- **Base-block** is a block of an array-base and maps directly into one block of memory located on one node. The memory block is contiguous and only one process has exclusive access to the block. The base-blocks are distributed across multiple processes in a round-robin fashion according to the N-Dimensional Block Cyclic Distribution.

- **View-block** is a block of an array-view and from the perspective of the user a view-block is a contiguous block of array elements. A view-block can span over multiple base-blocks and consequently also over multiple processes. For a process to access a whole view-block it will have to fetch data from possible remote processes and put the pieces together before accessing the block. To avoid this process, which may cause some internal memory copying, we divide view-blocks into sub-view-block.

- **Sub-view-block** is a block of data that is a part of a view-block but is located on only one process. The memory block is not necessarily contiguous but only one process has exclusive access to the block. The driving idea is that all array

operation is translated into a number of sub-view-block operations.

In this data model, an aligned array is an array that has a direct contiguous mapping through the block hierarchy. That is, a distributed array in which the base-blocks, view-blocks and sub-view-blocks are identical. A non-aligned array is then a distributed array without this property.

It is straightforward to parallelization aligned array operations because each view-block is identical to the underling base-block and is located on a single process. On the other hand, when operating on non-aligned arrays each view-block may be located on multiple processes. Therefore, we have to divide the computation into sub-view-blocks and even into aligned blocks of sub-view-blocks, which makes the operation more complex and introduces extra communication and computation overhead.

At the user level, an array operation operates on a number of input array-views and output array-views. It is the user's responsibility to make sure that the shape of these array-views matches each other. Since all arrays uses the same block size, this guaranties that all involved view-blocks match each other. Thus, it is possible to handle one view-block from each array at a time. In order to compute an array operation in parallel all available processes computes a view-block using the following steps:

1. The process fetches all the remote sub-view-blocks that constitute the involving input view-blocks.

2. The process aligns the sub-view-blocks by dividing them into the smaller blocks that are aligned to each other. If some output sub-view-blocks is not located on the process it will use temporary memory for the output.

3. The process applies operation on these aligned blocks.

4. The process sends temporary output sub-view-blocks back to the original locations.

7.3.3 3-Point Stencil Application

To demonstrate how the model works we will walkthrough the execution of the first block in a small 3-point stencil application. Two processes are executing the stencil application with the two array-bases, M and N, using a block-size of three elements. This means that three contiguous array elements are located on each process (Figure 7.9). The application uses two input array-views, A and B, and one output array-view, C, to compute the 3-point stencil.

In order to compute the first view-block in the three array-views, process 0 divides the computation into two parts (Figure 7.11). The first part, which consists of the first two elements, needs no communication since all elements are located locally. The process can therefore apply the operation directly on the first two elements of each array.

The second part, which consists of the third element, needs communication. The two processes will transfer the third element in A from process 1 to process 0. Even though the third element in C is located remotely, no communication is need now because C is

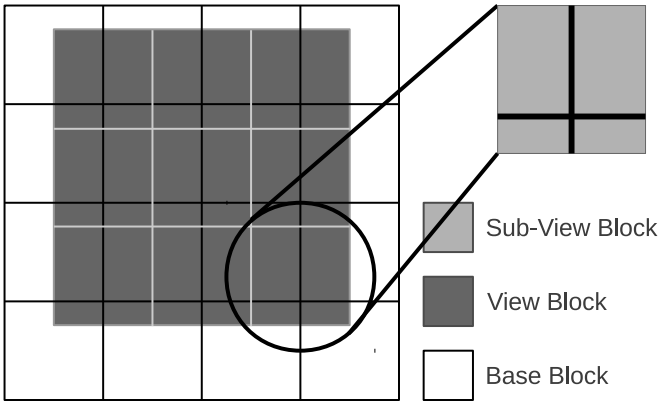


Figure 7.10: An illustration of the block hierarchy that represents a 2D distributed array. The array is divided into three block-types: Base, View and Sub-View-blocks. The 16 base-blocks make up the base-array, which may be distributed between multiple processes. The 9 view-blocks make up a view of the base-array and represent the elements that are visible to the user. Each view-block is furthermore divided into four sub-view-blocks, each located on a single process.

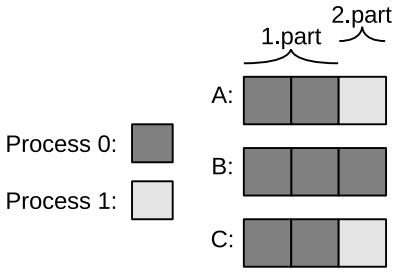


Figure 7.11: The sub-view-block alignment of the first view-block in the three array-views A, B and C (Figure 7.8 and 7.9).

the output. Instead, a temporary memory location is used for the output element. The process will apply the the operation when the communication the element is finished. When process 0 finishes the computation of part 2 the process transfer the third element back to process 1.

7.3.4 Latency-Hiding

It is essential to the performance of non-aligned array operations that the execution hides communication latency behind computation. In order to accomplish this, we make use of the Latency-Hiding model we will introduce in the next development stage of DistNumPy (Section 7.4).

Processor	AMD Opteron 6172
Clock	2.1 GHz
Peak Performance per Core	8.4 Gflops
Cores per NUMA Domain	6
NUMA Domains per Node	4 (packaged in 2 sockets)
Total Cores per Node	24
Private L1 Data Cache	64 KB
Private L2 Data Cache	512 KB
Shared L3 Cache per Socket	12MB
Memory Bandwidth	25.6 GB/s
Memory per Node	32GB DDR3-1066 ECC
Compiler	PGI 11.3
Math Library	Cray Scientific Library 10.5
Interconnect	Gemini 3-D Torus
Peak Bandwidth (per direction)	7 GB/s
MPI	Cray MPI 5.1.4

Table 7.2: Cray XE-6 Supercomputer

7.3.5 Experiments

In this section, we will evaluate the performance impact of our model for managing non-aligned array operations. We conduct all experiments on an Cray XE6 supercomputer (Table 7.2). The system systems consist of multi-core Non-Uniform Memory Access (NUMA) shared-memory nodes where each node has multiple NUMA domains. CPU cores within the same NUMA domain have uniform data access latency to the local memory while CPU cores of different NUMA domains would have non-uniform data access latencies. We will focus on the MPI communication overhead associated with non-aligned array operation and we will therefore only execute one MPI-process per NUMA domain.

To evaluate the performance, we will compare aligned array operations with non-aligned array operations. We use a 5-point stencil application that uses Jacobi Iteration in a fixed number of iterations. Figure 7.12 is this application implemented in Python using the DistNumPy library. It expresses the 5-point stencil using five array views that are shifted one element in each direction and thereby non-aligned operations (Figure 6.2). In order to benchmark the efficiency of the data structures hierarchy we introduce in this work, we compare this application with a synthetic version where all operations an aligned and do the same amount of computation. Because of the exclusively use of aligned operation the synthetic version requires no communication. It should be emphasize that the synthetic version is purely for benchmark purposes and do no meaningful work.

The unfavorable computation-communication ratio in the 5-point stencil application makes it difficult to achieve good scaling performance. The asymptotic computational complexity is $O(n)$ thus increasing the problem size does not improve the scaling performance significantly.

For the experiment, we calculate the FLOPS based on the floating operation counts of the ideal sequential algorithm and the measured execution times. Additionally, we compare the results with the linearly scaling performance, which we calculate by ex-

```
1  I #Number of iterations
2  A #Input & Output Matrix
3  SIZE //Symmetric Matrix Size
4  #Temporary array
5  T = empty([SIZE]*2,dtype=double,dist=True)
6  for i in xrange(I):
7    T[:] = A[1:-1, 1:-1] #Center
8    T   += A[1:-1, 0:-2] #Left
9    T   += A[1:-1, 2:  ] #Right
10   T   += A[0:-2, 1:-1] #Up
11   T   += A[2:  , 1:-1] #Down
12   A[1:-1, 1:-1] = T
```

Figure 7.12: 5-point stencil application that uses Jacobi Iteration in a fixed number of iterations implemented in DistNumPy

trapolating the sequential FLOPS performance of NumPy. We use this comparison as an upper bound of the achievable scalable performance. We perform weak scaling experiments, in which the problem size is scaled with the number of CPU-cores in the executions. The experiment goes from 8 to 2048 CPU-cores where the CPU-cores and problem size doubles between each execution.

Results

Figure 7.13 shows the result of the experiment. Overall the result is very promising, we see a linear increase of performance in both the aligned and non-aligned version. The aligned version demonstrates a speedup of 1514 at 2048 CPU-cores compared to a sequential execution, which translates into a CPU utilization of 74%. The non-aligned version demonstrates a speedup of 948 at 2048 CPU-cores compared to a sequential execution, which translates into a CPU utilization of 46%.

To analyze the experiment result further we divide the execution time into three categories in Figure 7.14. The execution time in each category is the average timing from each process.

Computation is the time used on actually computing element values. It should be fairly static through all the executions. However, variations in the data distribution may result in different execution times.

Blocking is the time used on waiting for communication to finish. Each process will do as much work as possible before interring a blocking state. However, as the number of CPU-cores increases the chances that the job scheduler on the Cray system allocates distant nodes to a job also increases. Furthermore, the torus network performance may suffer from the communication traffics caused by other jobs.

Overhead is the time used on handling the data structures associated with array oper-

ations. The overhead is proportional with the number of sub-view-blocks involved in the computation. Since the number of sub-view-blocks increases with the problem size, the overhead also increases. In addition, the number of sub-view-blocks increases even more when executing non-aligned operations.

As expected the blocking time is relatively small for all the aligned operation executions. Even at 2048, the blocking time is less the 2% of the total execution time. On the other hand, the blocking time for the non-aligned version is not as good. At 2048, the blocking time is 18% of the total execution time. This increase in blocking time is primarily because of an increase in communication, but also because of the MPI implementation by Cray. Currently, the Cray MPI for the Cray Gemini network has limited overlapping support for non-blocking MPI communication.

In the aligned operation version, the overhead time increases from 0.4% to 24% of the overall execution time. This overhead incensement is a direct result of the increased problem size. In the non-aligned operation version, the overhead increases more drastically – going from 6% to 34% of the overall execution time. This is because the non-aligned operations results in four times the number of sub-view-blocks – one sub-view-block per direction in the stencil computation.

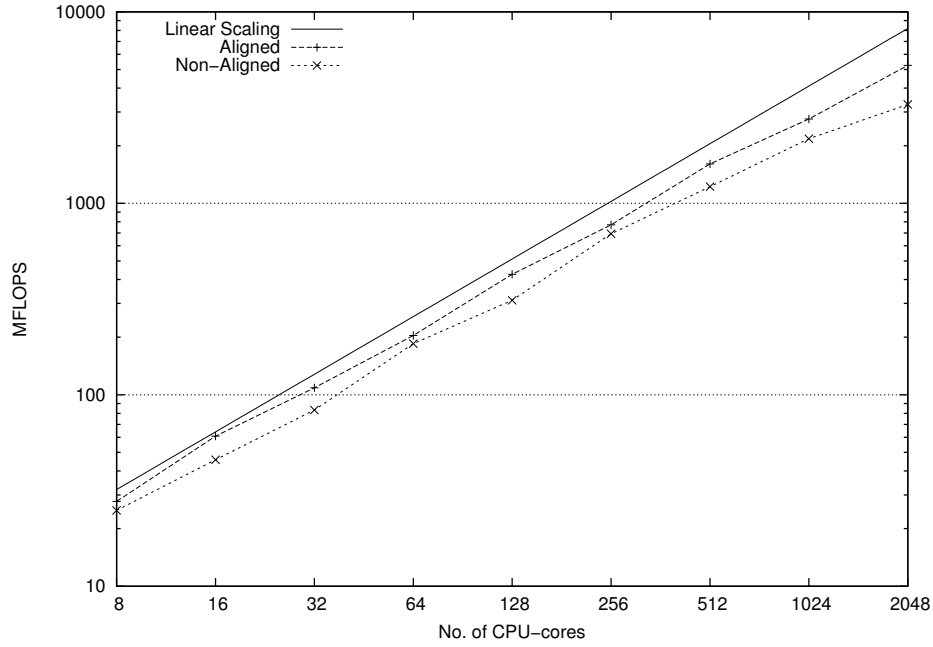


Figure 7.13: Weak scaling of aligned and non-aligned array operation.

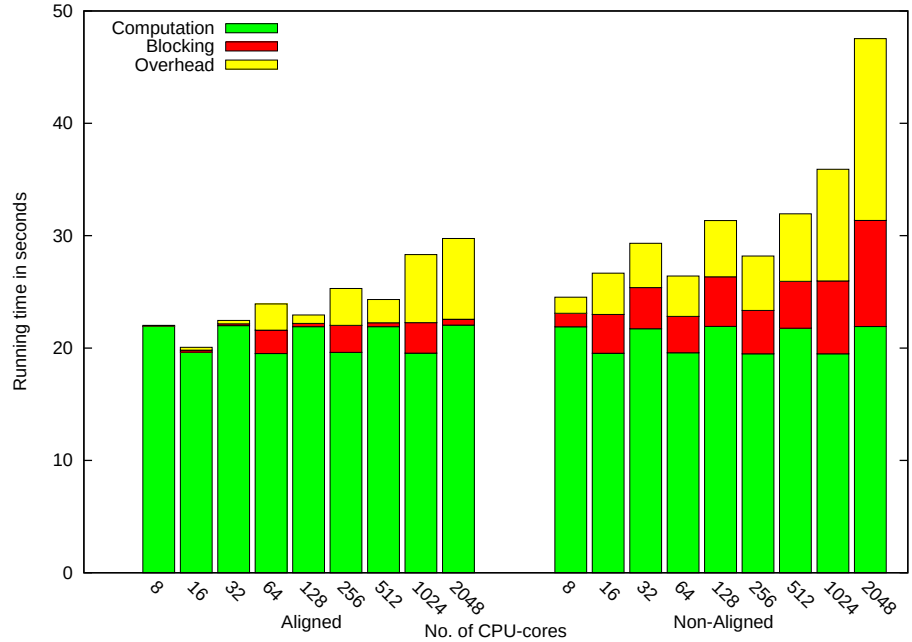


Figure 7.14: Weak scaling of aligned versus non-aligned array operation.

7.3.6 Conclusion

The single execution flow with abstract data operations is both the main strength and weakness of data-parallel programming models: two most notorious types of parallel programming bugs, data races and deadlocks, simply do not exist in data-parallel applications because there is only one execution thread. However, flexible abstract data operations for data-parallel applications require a very efficient runtime system in order to have good scalable performance.

In this work, we have successfully shown that by splitting data blocking based on locality is possible to efficiently managing abstract data structures that map to arbitrary distributed memory. We demonstrate scalable performance of a Jacobi Iteration application up to 2048 CPU-cores.

7.4 Communication Latency Hiding

The third development stage of DistNumPy introduces a runtime model for managing communication with support for latency-hiding. The model enables non-computer science researchers to exploit communication latency-hiding techniques seamlessly. For compiled languages, it is often possible to create efficient schedules for communication, but this is not the case for interpreted languages. By maintaining data dependencies between scheduled operations, it is possible to aggressively initiate communication and lazily evaluate tasks to allow maximal time for the communication to finish before entering a wait state. I implement a heuristic of this model in DistNumPy, an auto-parallelizing version of numerical Python that allows sequential NumPy programs to run on distributed memory architectures.

7.4.1 Introduction

To obtain performance in manual parallelization the programmer usually applies a technique known as latency-hiding, which is a well-known technique to improve the performance and scalability of communication bound problems and is mandatory in many scientific computations.

In this work, we introduce an abstract model to handle latency-hiding at runtime. The target is scientific applications that make use of vectorized computation. The model enables us to implement latency-hiding into high-productivity programming languages where the runtime system handles communication and parallelization exclusively.

In such high-productivity languages, a key feature is automatic distribution, parallelization and communication that are transparent to the user. Because of this transparency, the runtime system has to handle latency-hiding without any help. Furthermore, the runtime system has no knowledge of the communication pattern used by the user. A generic model for latency-hiding is therefore desirable.

The transparent latency-hiding enables a researcher that uses small self-maintained programs, to use a high-productivity programming language, Python in our case, without sacrificing the possibility of utilizing scalable distributed memory platforms. The

purpose of the work is not that the performance of an application, which is written in a high-productivity language, should compete with that of a manually parallelized compiled application. Rather the purpose is to close the gap between high-productivity on a single CPU and high performance on a parallel platform and thus have a high-productivity environment for scalable architectures.

The latency-hiding model proposed in this work is tailored to parallel programming languages and libraries with the following properties:

- The programming language requires dynamic scheduling at runtime because it is interpreted.
- The programming language supports and utilizes a distributed memory environment.
- All parallel processes have a global knowledge of the data distribution and computation.
- The programming language makes use of data parallelism in a Single Instruction, Multiple Data (SIMD) fashion in the sense that data affinity dictates the distribution of the computation.

Again, these properties match perfectly to DistNumPy but the latency-hiding model is general enough to support other parallel frameworks.

Latency-Hiding

We define latency-hiding informally as in [93] – "a technique to increase processor utilization by transferring data via the network while continuing with the computation at the same time". When implementing latency-hiding the overall performance depends on two issues: the ability of the communication layer to handle the communication asynchronously and the amount of computation that can overlap the communication – in this work we will focus on the latter issue.

In order to maximize the amount of communication hidden behind computation when performing vectorized computations our abstract latency-hiding model uses a greedy algorithm. The algorithm divides the arrays, and thereby the computation, into a number of fixed-sized data blocks. Since most numerical applications will work on identical dimensioned datasets, the distribution of the datasets will be identical. For many data blocks, the location will therefore be the same and these will be ready for execution without any data transfer. While the co-located data blocks are processed, the transfers of the data blocks from different location can be carried out in the background thus implementing latency-hiding. The performance of this algorithm relies on two properties:

- The number of data blocks must be significantly greater than number of parallel processors.
- A significant number of data blocks must share location.

In order to obtain both properties we need a data structure that support easy retrieval of dependencies between data blocks. Furthermore, the number of data blocks in a computation is proportional with the total problem size thus efficiency is of utmost importance.

Directed Acyclic Graph

It is well-known that a directed acyclic graph (DAG) can be used to express dependencies in parallel applications[4]. Nodes in the DAG represent operations and edges represent serialization dependencies between the operations, which in our case is due to conflicting data block accesses.

Scheduling operations in a DAG is a well-studied problem. The scheduling problem is NP-complete in its general forms [44] where operations are scheduled such that the overall computation time is minimized. There exist many heuristic for solving the scheduling problem [64], but none match our target.

The scheduling problem we solve in this work is not NP-hard because we are targeting programming frameworks that make use of data parallelism in a SIMD fashion. The parallel model we introduce is statically orchestrating data distribution and parallelization based on predefined data affinity. Assignment of computation tasks is not part of our scheduling problem. Instead, our scheduling problem consists of maximizing the amount of communication that overlaps computation when moving data to the process that is predefined to perform the computation.

In [90] the authors demonstrate that it is possible to dynamic schedule operations in a distributed environment using local DAGs. That is, each process runs a private runtime system and communicates with other processes regarding data dependences. Similarly, our scheduling problem is also dynamic but in our case all processes have a global knowledge of the data distribution and computation. Hence, no communication regarding data dependences is required at all.

The time complexity of insetting a node into a DAG, $G = (V, E)$, is $O(V)$ in worse case. Building the complete DAG is therefore $O(V^2)$. Removing one node from the DAG is $O(V)$, which means that in the case where we simply wants to schedule all operations in a legal order the time complexity is $O(V^2)$. This is without minimizing the overall computation or the amount of communication hidden behind computation. We therefore conclude that a complete DAG approach is inadequate for runtime control of latency-hiding in our case.

We address the shortcoming of the DAG approach through a heuristic that manage dependencies on individual blocks. Instead of having a complete DAG, we maintain a list of depending operations for each data block. Still, the time complexity of scheduling all operations is $O(V^2)$ in worse case, but the heuristic exploits the observation that in the common case a scientific application spreads a vectorized operation evenly between the data blocks in the involved arrays. Thus the number of dependencies associated with a single data block is manageable by a simple linked list. In Section 7.4.2, we will present a practical implementation of the idea.

Universal Function

A universal function (ufunc) is an element-wise vector operation that computes all elements in an array-view independently. Applying an ufunc operation on a whole array-view is semantically equivalent to performing the ufunc operation on each array-view block individually. This property makes it possible to perform a distributed ufunc operation in parallel. A distributed ufunc operation consists of four steps:

1. All MPI-processes determine the distribution of the view-block computation, which is strictly based on the distribution of the output array-view.
2. All MPI-processes exchange array elements in such a manner that each MPI-process can perform its computation locally.
3. All MPI-processes perform their local computation.
4. All MPI-processes send the altered array elements back to the original locations.

7.4.2 Latency-Hiding

The standard approach to hide communication latency behind computation in message-passing is a technique known as double buffering. The implementation of double buffering is straightforward when operating on a set of data block that all have identical sizes. The communication of one data block is overlapped with the computation of another already communicated data block and since the sizes of all the data blocks are identical all iterations are identical.

In DistNumPy, a straightforward double buffering approach works well for ufuncs that operate on aligned arrays, because it translates into communication and computation operations on whole view-blocks, which does not benefit from latency-hiding inside view-blocks. However, for ufuncs that operate on non-aligned arrays this is not the case because the view-block is distributed between multiple MPI-processes. In order to achieve good scalable performance the DistNumPy implementation must therefore introduce latency-hiding inside view-blocks. For example the computation of a view-block in Figure 7.10 can make use of latency-hiding by first initiating the communication of the three non-local sub-view-blocks then compute the local sub-view-block and finally compute the three communicated sub-view-blocks.

Operation Dependencies

One of the key contributions in this work is a latency-hiding model that, by maintaining data dependencies between scheduled operations, is able to aggressively initiate communication and lazily evaluate tasks, in order to allow maximal time for the communication to finish before entering a wait state. In this section, we will demonstrate the idea of the model by giving an example of a small 3-point stencil computation implemented in DistNumPy (Figure 7.8). For now, we will use a traditional DAG for handling the data dependencies. Later we will describe the implementation of the heuristic that enables

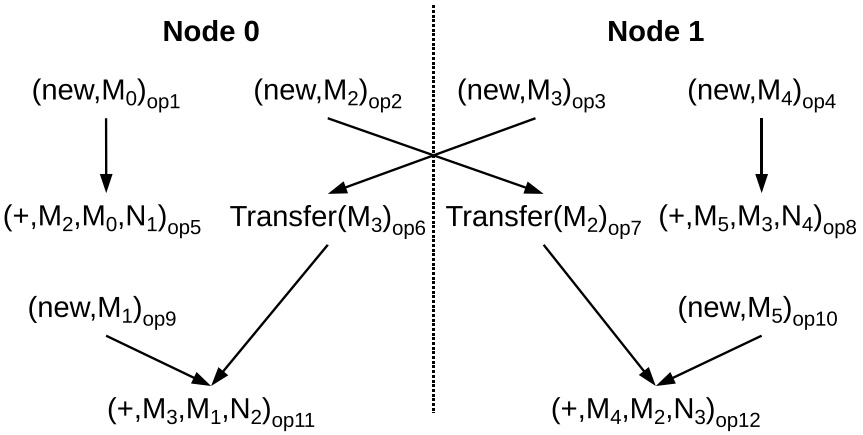


Figure 7.15: Illustration of a DAG that represents the dependencies in a 3-point stencil application (Figure 7.8). The DAG consists of 12 operations, *op1 to op12*, divided between two processes.

us to manage dependencies more efficiently. Additionally, it should be noted that the parallel processes do not need to exchange dependency information since they all have full knowledge of the data distribution.

Two processes are executing the stencil application and DistNumPy distributes the two arrays, M and N, using a block-size of three. This means that three contiguous array elements are located on each process (Figure 7.9). Using a DAG as defined in section 7.4.1, Figure 7.15 illustrates the dependencies between 12 operations that together constitute the execution. Initially the following six operations are ready:

$$R := \{op1, op2, op3, op4, op9, op10\}$$

Afterwards, without the need of communication, two more operations $op5$ and $op8$ may be executed. Thus, it is possible to introduce latency-hiding by initiating the communication, $op6$ and $op7$, before evaluating operation $op5$ and $op8$. The amount of communication latency hidden depends on the computation time of $op5$ and $op8$ and the communication time of $op6$ and $op7$.

We will strictly prioritize between operations based on whether they involve communication or computation – giving priority to communication over computation. Furthermore, we will assume that all operations take the same amount of time, which is a reasonable assumption in DistNumPy since it divides array operations into small blocks that often have the same computation or communication time.

Lazy Evaluation

Since Python is an interpreted dynamic programming language, it is not possible to schedule communication and computation operations at compile time. Instead, we introduce lazy evaluation as a technique to determine the communication and computation operations used in the program at runtime.

During the execution of a DistNumPy program all MPI-processes record the requested array operations rather than applying them immediately. The MPI-processes maintain the operations in a convenient data structure and at a later point all MPI-processes apply the operations. The idea is that by having a set of operations to carry out it may be possible to schedule communication and computation operations that have no mutual dependencies in parallel.

We will only introduce lazy evaluation for Python operations that involve distributed arrays. If the Python interpreter encounters operations that do not include DistNumPy arrays, the interpreter will execute them immediately. At some point, the Python interpreter will trigger DistNumPy to execute all previously recorded operation. This mechanism of executing all recorded operation we will call an *operation flush* and the following three conditions may trigger it.

- The Python interpreter issues a read from distributed data. E.g. when the interpreter reaches a branch statement.
- The number of delayed operations reaches a user-defined threshold.
- The Python interpreter reaches the end of the program.

The Dependency System

The main challenge when introducing lazy evaluation is to implement a dependency system that schedules operations in a performance efficient manner while the implementation keeps the overhead at an acceptable level.

Our first lazy evaluation approach makes use of a DAG-based data structure to contain all recorded operations. When an operation is recorded, it is split across the sub-view-blocks that are involved in the operation. For each such operation, a DAG node is created just as in Figure 7.8 and 7.9.

Beside the DAG our dependency system also consist of a *ready queue*, which is a queue of recorded operations that do not have any dependencies. The ready queue makes it possible to find operations that are ready to be executed in the time complexity of $O(1)$.

Operation Insertion The recording of an operation triggers an insertion of new node into the DAG. A straightforward approach will simply implement insertion by comparing the new node with all the nodes already located in the DAG. If a dependency is detected the implementation adds an edge between the nodes. The time complexity of such an implementation is $O(n)$ where n is the number of operation in the DAG and the construction of the complete DAG is $O(n^2)$.

Operation Flush To achieve good performance the operation flush implementation must maximize the amount of communication that it is overlapped by computation. Therefore, the flush implementation initiate communication at the earliest point in time and only do computation when all communication has been initiated. Furthermore, to make sure that there is progress in the MPI layer it checks for finished communication

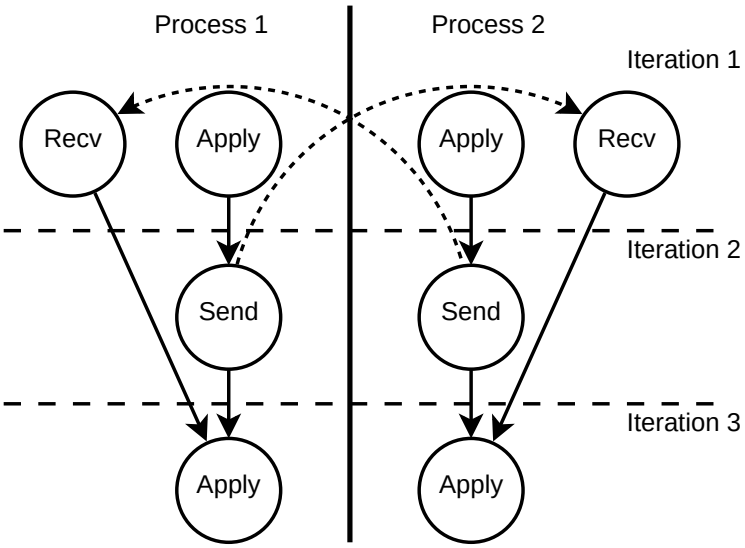


Figure 7.16: Illustration of the naïve evaluation approach. The result is a deadlock in the first iteration since both processes are waiting for the receive-node to finish, but that will never happen because the matching send-node is in second iteration.

in between multiple computation operations. The following is the flow of our operation flush algorithm:

1. Initiate all communication operations in the ready queue.
2. Check in a non-blocking manner if some communication operations have finished and remove finished communication operations from the ready queue and the DAG. Furthermore, register operations that now have no dependencies into the ready queue.
3. If there is only computation operations in the ready queue execute one of them and remove it from the ready queue and the DAG.
4. Go back to step one if there are any operations left in the ready queue else we are finished.

The algorithm maintains the following three invariants:

1. All operations, that are ready, are located in the ready queue.
2. We only start the execution of a computation node when there is no communication node in the ready queue.
3. We only wait for communication when the ready queue has no computation nodes.

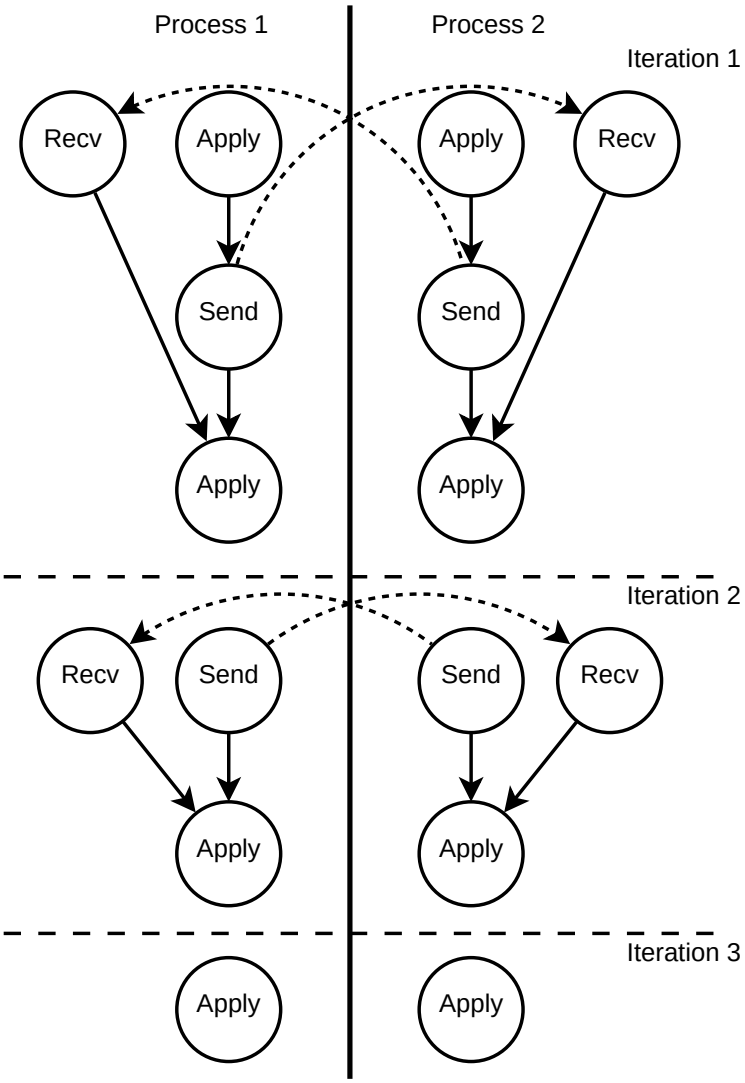


Figure 7.17: Illustration of a deadlock-free evaluation of the dependency graph in Figure 7.16. Each MPI-process evaluates as much as possible before waiting for any communication.

Deadlocks To avoid deadlocks a MPI-process will only enter a blocking state when it has initiated all communication and finished all ready computation. This guaranties a deadlock-free execution but it also reduces the flexibility of the execution order. Still, it is possible to check for finished communication using non-blocking functions such as `MPI_Testsome()`.

The naïve approach to evaluate a DAG is simply to first evaluate all nodes that have no dependencies and then remove the evaluated nodes from the graph and start over – similar to the traditional BSP model. However, this approach may result in a deadlock as illustrated in Figure 7.16. Figure 7.17 illustrate the same DAG executed using our approach where a MPI-process evaluates as much as possible before entering a blocking state.

Dependency Heuristic Experiments with lazy evaluation using the DAG-based data structure shows that the overhead associated with the creation of the DAG is very time consuming and becomes the dominating performance factor. We therefore introduce a heuristic to speed up the common case. We base the heuristic on the following two observations:

- In the common case, a scientific DistNumPy application spreads a computation evenly between all sub-view-blocks in the involved arrays.
- Operation dependencies are only possible between sub-view-blocks that are part of the same base-block.

The heuristic is that instead of having a DAG, we introduce a prioritized operation list for each base-block. The assumption is that, in the common case, the number of operations associated with a base-block is manageable by a linked list.

We implement the heuristic using the following algorithm. A number of operation-nodes and access-nodes represent the operation dependencies. The operation-node contains all information needed to execute the operation on a set of sub-view-blocks and there is a pointer to an access-node for each sub-view-block. The access-node represents memory access to a sub-view-block, which can be either reading or writing. E.g., the representation of an addition operation on three sub-view-blocks is two read access-nodes and one write access-node (Figure 7.18).

Our algorithm places all access-nodes in dependency-lists based on the base-block that they are accessing. When an operation-node is recorded each associated access-node is inserted into the dependency list of the sub-view-blocks they access. Additionally, the number of accumulated dependencies the access-nodes encounter is saved as the operation-node's reference counter.

All operation-nodes that are ready for execution have a reference count of zero and are in the ready queue. Still, they may have references to access-nodes in dependency-lists – only when we execute an operation-node will we remove the associated access-nodes from the dependency-lists. Following the removal of an access-node we traverse the dependency-list and for each depending access-node we reduce the associating reference

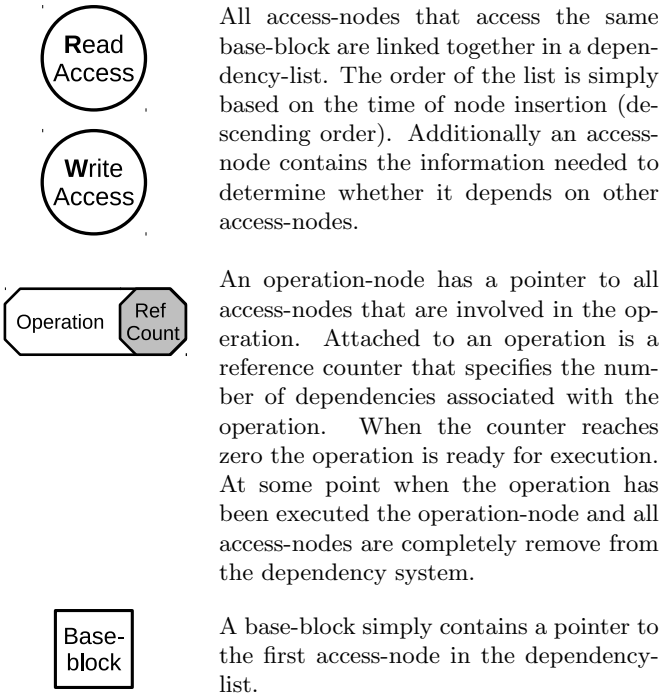


Figure 7.18: The structures used in the dependency system.

counter by one. Because of this, the reference counter of another operation-node may be reduced to zero, in which case we move the operation-node to the ready queue and the algorithm starts all over.

Figure 7.18 goes through all the structures that make up the dependency system and Figure 7.19 illustrates a snapshot of the dependency system when executing the 3-point stencil application.

Table 7.3: Hardware Specifications

CPU	Intel Xeon E5345
CPU Frequency	2.33 GHz
CPU per node	2
Cores per CPU	4
Memory per node	16 GB
Number of nodes	16
Interconnect	Gigabit Ethernet
Compiler	GCC 4.4.3
MPI	Open MPI 1.5.1

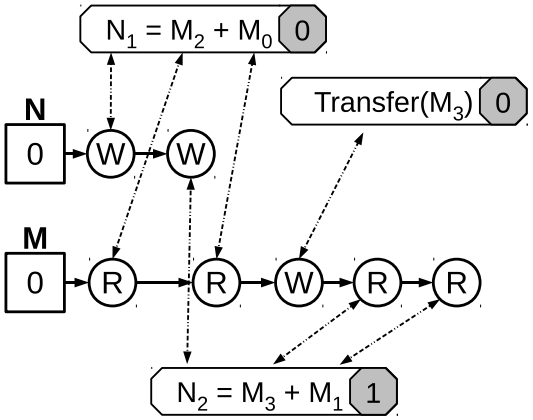


Figure 7.19: Illustration of the dependency system when executing the 3-point stencil in Figure 7.8, 7.9 and 7.15. The illustration is a snapshot of the dependency system on node 0 after the creation of all the arrays. Note that since the block size is three, node 0 only has one block of each array.

7.4.3 Experiments

To evaluate the performance impact of the latency-hiding model introduced in this work, we will conduct performance benchmark using DistNumPy and NumPy[3]. The benchmark is executed on an Intel Core 2 Quad cluster (Table 7.3) and for each application we calculate the speedup of DistNumPy compared to NumPy. The problem size is constant though all the executions, i.e. we are measuring strong scaling. To measure the performance impact of the latency-hiding, we use two different setups: one with latency-hiding and one that uses blocking communication. For both setups we measured the time spent on waiting for communication, i.e. the communication latency not hidden behind computation.

In this benchmark we utilizes the cluster in a *by node* fashion. That is, from one to sixteen CPU-cores we start on MPI-process per node (no SMP) and above sixteen CPU-cores we start multiple MPI-processes per node. The MPI library used throughout this benchmark is OpenMPI[4].

The benchmark consists of the following eight Python applications.

- **Fractal** Computation of the Mandelbrot Set. From a NumPy tutorial written by Walt[98] (Figure 7.22).
- **Black-Scholes** Computation of the Black-Scholes model[14] implemented in NumPy (Figure 7.20 and 7.23).

  Both **Fractal** and **Black-Scholes** are embarrassingly parallel applications and we expect that latency-hiding will not improve the performance.

[3]NumPy version 1.3.0
[4]OpenMPI version 1.5.1

- **N-body** A Newtonian N-body simulation that uses a naïve algorithm that computes all body-body interactions. The NumPy implementation is a translation of a MATLAB application by Casanova[26] (Figure 7.24).

- **kNN** A naïve implementation of a k nearest neighbor search (Figure 7.25).

  The **N-body** and **kNN** applications have a computation complexity of $O(n^2)$. This indicates that the two applications should have good scalability even without latency-hiding.

- **Lattice Boltzmann 2D** Lattice Boltzmann model of channel flow in 2D using the D2Q9 model. It is a translation of a MATLAB application by Latt[70] (Figure 7.26).

- **Lattice Boltzmann 3D** Lattice Boltzmann model of a fluid in 3D using the D3Q19 model. It is a translation of a MATLAB application by Haslam[53] (Figure 7.27).

  The two **Lattice Boltzmann** applications have a computation complexity of $O(n)$. More communication is therefore needed and we expect that latency-hiding will improve the performance.

- **Jacobi** The Jacobi method is an algorithm for determining the solutions of a system of linear equations. It is an iterative method that uses a spitting scheme to approximate the result (Figure 7.28).

- **Jacobi Stencil** In this benchmark, we have implemented the Jacobi method using stencil operations rather than matrix row operations (Figure 7.21 and 7.29).

  The two **Jacobi** applications also have a computation complexity of $O(n)$. However, the constant associated with n is very small, e.g. to compute one element in the Jacob Stencil application four adjacent elements are required. We expect latency-hiding to be very important for good scalability.

Discussion

Overall, the benchmarks show that DistNumPy has acceptable performance and scalability. However, the scalability is somewhat worsening at 32 CPU-cores and above, which correlates with the use of multiple CPU-cores per node. Because of this distinct performance profile, we separate the following discussion into results executed on one to sixteen CPU-cores (one CPU-core per node) and the results executed on 32 CPU-cores to 128 CPU-cores (multiple CPU-cores per node).

One to Sixteen CPU-cores The benchmarks clearly shows that DistNumPy has both good performance and scalability. Actually, half of the Python applications achieve super-linear speedup at sixteen CPU-cores. This is possible because DistNumPy, opposed to NumPy, will try to reuse memory allocations by lazily de-allocating arrays.

```
# Black Scholes Function
#  S: Stock price
#  X: Strike price
#  T: Years to maturity
#  r: Risk-free rate
#  v: Volatility
def BlackScholes(CallPutFlag,S,X,T,r,v):
 d1 = (log(S/X)+(r+v*v/2.)*T)/(v*sqrt(T))
 d2 = d1-v*sqrt(T)
 if CallPutFlag=='c':
   return S*CND(d1)-X*exp(-r*T)*CND(d2)
 else:
   return X*exp(-r*T)*CND(-d2)-S*CND(-d1)
```

Figure 7.20: This is the Black Sholes Function in the **Black-Scholes** benchmark where `CND` is the cumulative normal distribution. Note that there is no source code difference between a parallel and a sequential version – it is regular Python/Numpy source code.

```
while epsilon < delta:
  T = 0.2 * (A[1:-1,1:-1] + A[2:,1:-1] \
           + A[1:-1,0:-2] + A[1:-1,2:])
  delta = sum(abs(A[1:-1,1:-1] - T))
  A[1:-1,1:-1] = T
```

Figure 7.21: This is the kernel in the **Jacobi Stencil** benchmark. Note that there is no source code difference between a parallel and a sequential version – it is regular Python/Numpy source code.

DistNumPy uses a very naïve algorithm that simply checks if a new array allocation is identical to a just de-allocated array. If that is the case one memory allocation and de-allocation is avoided.

In the two embarrassingly parallel applications, **Fractal** and **Black-Scholes**, we see very good speedup as expected. Since the use of communication in both applications is almost non-existing the latency-hiding makes no difference. The speedup achieved at sixteen CPU-cores is 18.8 and 15.4, respectively.

The two naïve implementations of **N-body** and **kNN** do not benefit from latency-hiding. In **N-body** the dominating operations are matrix-multiplications, which is a native operation in NymPy and in DistNumPy implemented as specialized operations using the parallel algorithm SUMMA[47] and not as a combination of ufunc operations. Since both the latency-hiding and the blocking execution use the same SUMMA algorithm the performance is very similar. However, because of the overhead associated with latency-hiding, the blocking execution performs a bit better. The speedup achieved at sixteen CPU-cores is 17.2 with latency-hiding and 17.8 with blocking execution. Similarly, the performance difference between latency-hiding and blocking in **kNN** is minimal – the speedup achieved at sixteen CPU-cores is 12.5 and 12.6, respectively. The relatively modest speedup in **kNN** is the result of poor load balancing. At eight and sixteen CPU-cores the chosen problem is not divided evenly between the processes.

Latency-hiding is somewhat beneficial to the two **Lattice Boltzmann** applications. The waiting time percentage on sixteen CPU-cores goes from 19% to 13% in **Lattice Boltzmann 2D**, and from 16% to 9% in **Lattice Boltzmann 3D**. However, the overall impact on the speedup is not that great, primarily because of the overhead associated with latency-hiding.

Finally, latency-hiding introduces a drastically improved performance to the two communication intensive applications **Jacobi** and **Jacobi Stencil**. The waiting time percentage going from 54% to 2% and from 62% to 9%, respectively. Latency-hiding also has a major impact on the overall speedup, which goes from 5.9 to 12.8 and from 7.7 to 18.4, respectively. In other words latency-hiding more than doubles the overall speedup and CPU utilization.

Scaling above sixteen CPU-cores Overall, the performance is worsening at 32 CPU-cores – particular at 64 CPU-cores and above where the CPU utilization is below 50%. One reason for this performance degradation is the characteristic of strong scaling. In order to have considerable more data blocks than MPI-processes, the size of the data distribution blocks decreases as the number of executing CPU-cores increases. Smaller data blocks reduces the performance since the overhead in DistNumPy is proportional with the size of a data block.

However, smaller data blocks are not enough to justify the observed performance degradation. The von Neumann bottleneck[8] associated with main memory also hinder scalability. This is evident when looking at Figure 7.30, which is a speedup graph of **N-body** that compares *by node* and *by core* scaling. At eight CPU-cores, both result uses identical data distribution and block size, but the performance when only using one

CPU-core per node is clearly superior to using all eight CPU-cores on one node.

A NumPy application will often use ufuncs heavily, which makes the application vulnerable to the von Neumann bottleneck. The problem is that multiple ufunc operations are not pipelined in order to utilize cache locality. Instead, NumPy will compute a single ufunc operation at a time. This problem is also evident in DistNumPy since our latency-hiding model only address communication latency and not memory latency.

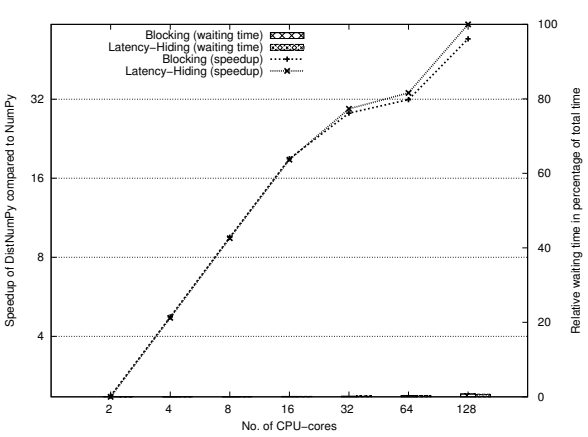


Figure 7.22: Speedup of the Fractal application.

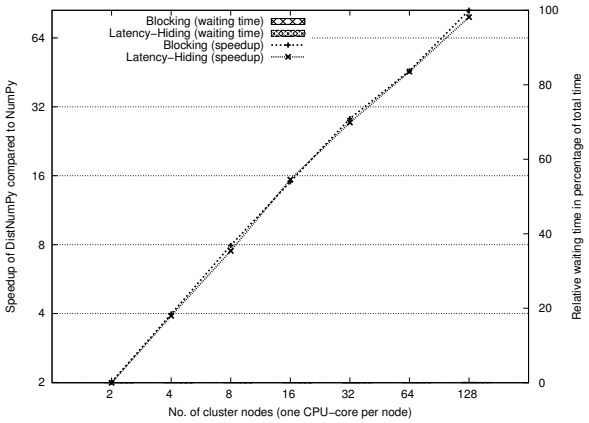


Figure 7.23: Speedup of the Black-Scholes application.

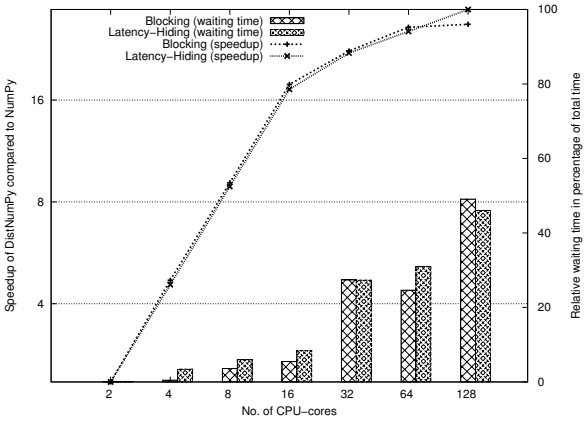


Figure 7.24: Speedup of the N-body application.

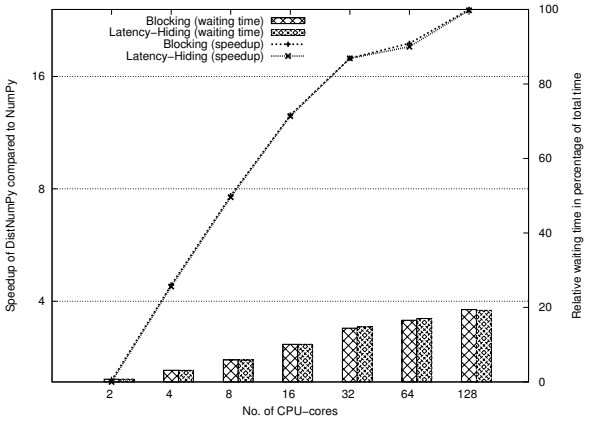


Figure 7.25: Speedup of the kNN application.

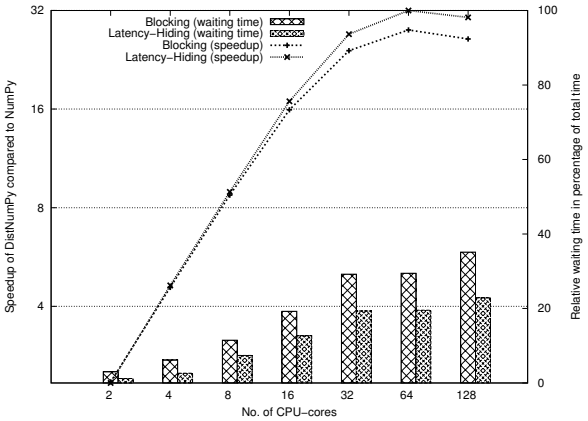


Figure 7.26: Speedup of the Lattice Boltzmann 2D application.

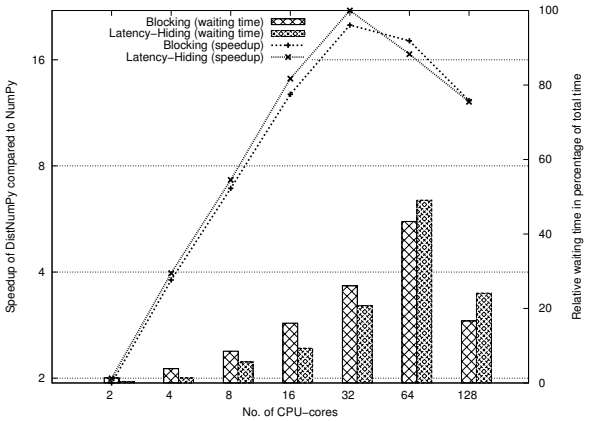


Figure 7.27: Speedup of the Lattice Boltzmann 3D application.

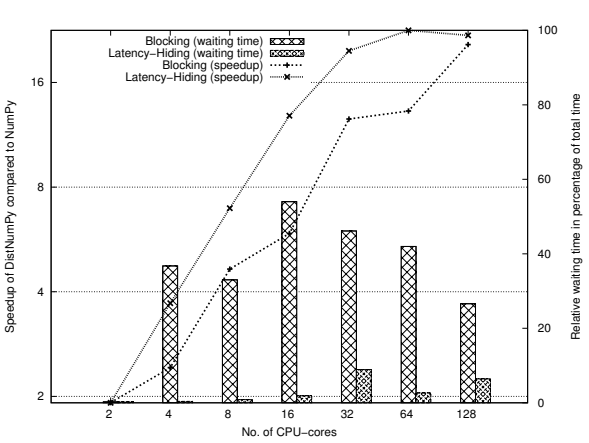


Figure 7.28: Speedup of the Jacobi application.

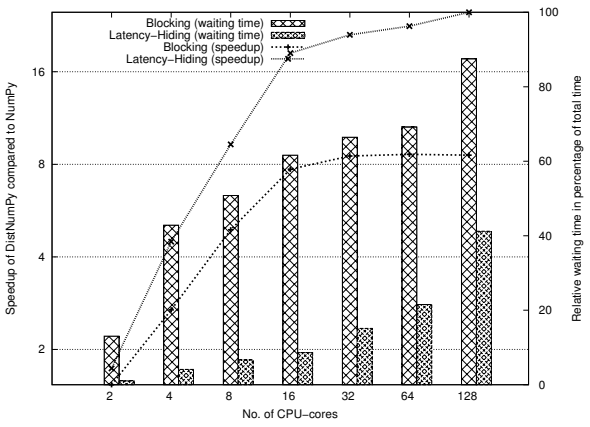


Figure 7.29: Speedup of the Jacobi Stencil application.

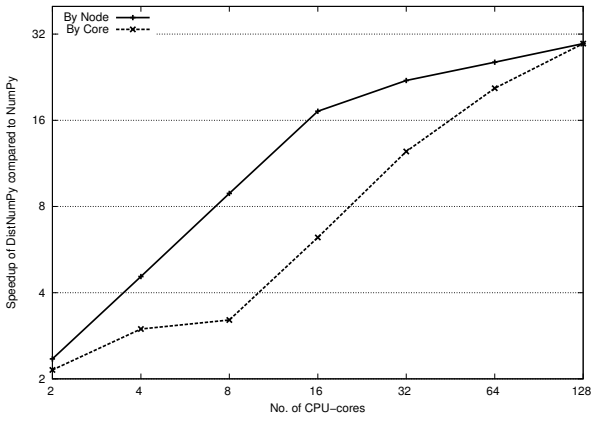


Figure 7.30: Speedup of the N-body application that compares *by node*, in which the maximum number of CPU-cores is used, and *by core*, in which the minimum number of nodes is used. Note that the *by node* graph is identical to the *latency-hiding* graph in Figure 7.24.

7.4.4 Conclusion

While automatic parallelization for distributed memory architectures cannot hope to compete with a manually parallelized version, the productivity that comes with automatic parallelization still makes the technique of interest to a user who only runs a code a few times between changes. For applications that are embarrassingly parallel or applications where the computational complexity is $O(n^2)$ or higher, it is fairly straight forward to manage the communication for automatic parallelization. However, for common kernels the complexity is $O(n \log(n))$ or even $O(n)$ and here the application of latency-hiding techniques is essential for performance.

In this work we have presented a scheme for managing latency-hiding, that is based on the assumption that splitting the work in more blocks than there are processors will allow us to aggressively communicate data-blocks between nodes, while at the same time processing operations that require no external data-blocks. The same dependency analysis may be done without a division into data-blocks, but the blocking approach allows us to maintain a full DAG, an operation that is known to be costly, and replace the DAG with a number of ordered linked lists, to which access is done in linear time.

We implement the model in Distributed Numerical Python, DistNumPy, a programming framework that allows linear algebra operations expressed in NumPy to be executed on distributed memory platforms and this is without any effort towards parallelization from the programmer.

A selection of eight benchmarks show that the system, as predicted, does not improve the performance of embarrassingly parallel applications or applications with complexity $O(n^2)$ or higher. For applications with lower complexity the benefit from automatic latency-hiding is highly dependent on the relationship between the amount of data that needs to be transferred and the cost of updating the individual elements.

In the Lattice Boltzmann codes, both 2D and 3D, the version without latency-hiding does quite well, simply because the operation of updating a data-point is time consuming enough to amortize the communication latency. However, when the cost of communication becomes more significant, as in a Jacobi solver and stencil-based Jacobi solver, the automatic latency-hiding significantly improve the performance.

The performance of the stencil-based Jacobi-solver improves from a speedup of 7.7 to 18.4 at sixteen processors and 8.6 to 25.0 at 128 processors, compared to standard sequential NumPy. This is matched by the fact that the time spend on waiting for communication drops from 62% to 9% and 87% to 41%, respectively, with the introduction of latency-hiding.

Overall, the conclusion is that managing latency-hiding at runtime is fully feasible and makes automatic parallelization feasible for a number of applications where manual parallelization would otherwise be required. The most obvious target is the large base of stencil-based algorithms.

7.5 PGAS-style Programming

The fourth and final development stage of DistNumPy introduces a parallel programming model that combines two well-known execution models: Single Instruction, Multiple Data (SIMD) and Single Program, Multiple Data (SPMD). The combined model supports SIMD-style data parallelism in global address space and supports SPMD-style task parallelism in local address space.

One of the most important features in the combined model is that data communication is expressed by global data assignments instead of message passing. I implemented this combined programming model into DistNumPy, making parallel programming with Python both highly productive and performing on distributed memory multi-core systems.

I implemented the SPMD task parallelism as an extension to DistNumPy that enables each process to have direct access to the local part of a shared array. To harvest the multi-core benefits in modern processors we exploit multi-threading in both SIMD and SPMD execution models. The multi-threading is completely transparent to the user – it is implemented in the runtime with OpenMP and by using multi-threaded libraries when available.

7.5.1 Introduction

There exists a broad range of execution models for parallel programming. Single Instruction, Multiple Data (SIMD) and Single Program, Multiple Data (SPMD) are two execution models often used in parallel programming. Our SIMD execution model refers to a single sequential Python instruction stream with massive data parallelism. Our programming model is executed in SIMD mode from the user's perspective but the underlying runtime system is built on top on MPI and is executed in SPMD mode.

SPMD Extension to DistNumPy

In this work, we introduce an extension to DistNumPy that mixes the already existing SIMD execution model in DistNumPy with the SPMD execution model. This new extension enables users to express parallel algorithms in terms of global data management and local operations. Using LINPACK as an example, the user may express the distributed-memory LU factorization algorithm in Python and use BLAS/LAPACK for local computations.

To overcome the relatively slow execution of the Python interpreter, we use four optimization techniques to amortize the overheads:

- Multi-threading with OpenMP to exploit data parallelism in array operations.
- Use optimized libraries for basic local computations whenever possible, such as BLAS, LAPACK, FFTW and vendor-optimized libraries. In common cases, most execution time would be spent on computation in the library and thus the overheads incurred by the Python interpreter are negligible. Even for applications

written in FORTRAN and C, it is a good practice to use optimized libraries if available because they usually run much faster than the standard implementations.

- Combine array operations through lazy evaluation. Using an internal dependency tracking system, the DistNumPy runtime system will aggregate operations and execute them in batch only when the results are required in the data flow. This lazy evaluation strategy can not only perform code optimization on-the-fly but also minimize the overheads of executing individual Python instructions.

- Overlap computation and communication by leveraging non-blocking communication. With sufficient overlapping, the Python interpreter overhead is hidden and will not increase the execution time.

In case all fails, the user still has the option to implement the performance-critical section in low-level languages and use it in the Python code to speed up execution. Because the Python implementation is more concise to understand, it is much easier to identify the bottleneck in such program than searching it in a very large C or FORTRAN application.

- Added support for SIMD and SPMD execution models in a single parallel program so that both data-parallel and task-parallel applications can be conveniently implemented with Python.

- Improved parallelism and scalability of DistNumPy by implementing hierarchical parallelism in the runtime: using OpenMP for multi-threading and MPI for inter-node communication.

- Enabled interoperability between DistNumPy and existing third-party libraries for efficient local computation.

- Developed three benchmarks with our proposed SIMD+SPMD programming model and evaluated their performance on up to 1536 cores.

7.5.2 Programming model

We propose a new PGAS programming model that combines the SIMD execution model for global array operations and the SPMD execution model for local array operations.

Global Array Operations

DistNumPy supports global arrays distributed among all available processes. Python applications can make use of DistNumPy by creating such global arrays. All global array operations use the SIMD execution model, in which all processes need to execute the same Python statement sequence even if some of them don't need to take actions. The computation place is based on the "owner computes" rule. Processes that own part or all the operands of an operation need to participate in the operation. Non-participating processes would simply mask out the current Python statement.

```
1  for i in xrange(nrow / BS):
2    #Apply local matrix multiplication
3    C.local()[:] += np.dot(A.local(), B.local())
4
5    #Moving columns horizontal (left)
6    tmp = A[:,0:BS].copy()
7    A[:,0:-BS] = A[:,BS:]
8    A[:,-BS:] = tmp
9
10   #Moving rows vertical (up)
11   tmp = B[0:BS,:].copy()
12   B[0:-BS,:] = B[BS:,:]
13   B[-BS:,:] = tmp
```

Figure 7.31: Cannon's algorithm

In DistNumPy, each process runs a Python interpreter that interprets the Python application. However, because of the synchronous nature of SIMD, the parallelism provided by DistNumPy is completely transparent to the user when every interpreter takes the same code paths and only uses global arrays operation. This transparency enables a user familiar with Python/NumPy to utilize multiple processes seamlessly. Figure 7.21 shows the kernel of a 5-point-stencil DistNumPy application where all arrays are global and all participating Python interpreters will execute identical code. The parallel code is identical to the sequential NumPy version and the data-parallel operations are automatically executed in parallel on distributed-memory computers.

Global Assignment Operations

The SPMD execution model typically uses explicit message passing (one or two sided) to move data in a distributed environment. DistNumPy enables using global data assignments to express data movement. For example, when multiplying two matrices by Cannon's algorithm, we shift matrix-blocks in the vertical and horizontal directions. Figure 7.31 line 5-13, is an implementation of this data movement using regular Python assignments. Actual communication associated with the assignments is completely hidden to the user. The user only needs to specify the algorithm's data dependencies and DistNumPy handles how to perform the data movements.

Local Array Operations

The user can get the local part of a global array from DistNumPy by the `local` function, which is the only global array operation that is non-collective and does not imply synchronization. The `local` function returns a NumPy array view that can be used together with any NumPy compatible Python libraries.

Figure 7.31 is an implementation of Cannon's algorithm where we assume the matrices are square. The implementation makes use of the local address space to compute

the local matrix multiplication (line 3) and the global address space to move matrix blocks up and left (line 5-13). The user can mix local and global operations as needed. The only rule is that a local array becomes undefined when executing a global array operation. The user needs to use `local` to retrieve a new local NumPy array after a global array operation . This restriction is because of the lazy evaluation used in the original DistNumPy (Section 7.5.3).

Local Array Block Iterator

Iterating over all local array blocks is a common operation. `blocks` is an operation that returns such an iterator. It is semantically equivalent to `local` but instead of returning the local part of the global array, it returns an iterator that iterates over all local blocks in the global array.

Data Layout

The user may have to be aware of the data layout when using local array operations. In the SPMD execution model, the user will often have to differentiate computations based on the process identity and data layout. To facilitate this, we assign a rank id to each process that is accessible through the constant `RANK`. The user may specify the data layout for global arrays at job start-up time and it is immutable over the course of the execution.

DistNumPy global arrays are stored in generalized N-Dimensional block cyclic distribution schemes inspired by HPF [42]. The user can define flexible process grids by using the `datalayout()` function. The block size is the same for all global arrays and is accessible through the constant `BLOCKSIZE`.

One Process Distribution

DistNumPy supports an alternative array distribution where all data is located on a single process. When creating a distributed array it is possible to specify where the data is located by a rank affinity parameter. The following operation will create a global array that is located on process 42 exclusively:

```
A = np.array([1,2,3],dist=True,onenode=42)
```

7.5.3 Implementation

DistNumPy is extended from NumPy as a Python package that can be used with regular Python interpreters. DistNumPy uses dynamic data dependency analysis, lazy evaluation and communication aggregation techniques to hide communication latency. Following the data dependencies between batched operations, DistNumPy proactively initiates data transfers as early possible while consumes the data as late as possible to maximize overlapping between communication and computation.

Multi-threading with OpenMP

The lack of efficient multi-threading support in the original DistNumPy is a severe limitation when executing on multi-core distributed memory systems. We improve the performance for data-parallel array operations by using multi-threading with OpenMP.

In NumPy and DistNumPy, a universal function (ufunc) is a vectorized function that operates on all array elements independently and provides implicit data-parallelism. In the runtime, we use OpenMP directives to parallelize the *for* loops in the ufunc computations.

The current multi-threading implementation harvests parallelism within single operation instead of parallelism over multiple operations because we find that it requires extensive dependency analyses to do so, which is beyond the scope of this work.

Third Party Python Libraries

There exist a great number of optimized numerical Python libraries. Most of them are compatible with NumPy because NumPy provides a C-pointer to the raw array data. DistNumPy can also make use of these NumPy libraries for local computations by converting the local part of the global array to a regular NumPy array. For example, in the Cannon's algorithm we use the local NumPy function `dot()` (Figure 7.31, line 3), which is a simple binding to an optimized BLAS library, to compute local matrix multiplication.

7.5.4 Benchmarks

To evaluate the implementation of our proposed programming model, we developed three mini-benchmarks: 1) matrix multiplication, 2) LU factorization and 3) 2-D heat equation solution with the Jacobi iterative method. We implemented all three benchmarks in pure Python and then used third party libraries, BLAS and LAPACK, to compute local results when applicable.

Matrix Multiplication

Matrix multiplication is a fundamental operation in numerical computations. The global matrices are distributed across all nodes with 2-D block-cyclic date layout. We use the SUMMA [47] algorithm, which is based on outer-product BLAS level-3 updates implemented by row and column broadcast communication followed by local matrix multiplications on each node.

Figure 7.32 shows the complete source code of the SUMMA implementation. It is completely written in Python and uses the NumPy function `np.dot()` (line 26) to compute the local matrix multiplication. `np.dot()` calls the optimized BLAS library available on the system.

The only communication in the SUMMA algorithm is in line 15-19, in which we replicate a column-block horizontally and a row-block vertically. The communication is elegantly expressed by global array assignments to `a_work` and `b_work`.

```
1  import numpy as np
2
3  def summa(a,b,c):
4    (prow,pcol) = a.pgrid()
5    BS = np.BLOCKSIZE
6    a_work = np.zeros((a.shape[0],BS*pcol), \
7                      dtype=a.dtype, dist=True)
8    b_work = np.zeros((BS*prow,b.shape[1]), \
9                      dtype=a.dtype, dist=True)
10   Ksz = a.shape[1]
11   for k in xrange(0,Ksz,BS):
12     bs = min(BS, Ksz-k)#Current block size
13
14     #Replicate column-block horizontal
15     for p in xrange(pcol):
16       a_work[:,p*BS:p*BS+bs] = a[:,k:k+bs]
17     #Replicate row-block vertical
18     for p in xrange(prow):
19       b_work[p*BS:p*BS+bs,:] = b[k:k+bs,:]
20
21     #Apply local outer dot product
22     l_a_work = a_work.local()[:,:bs]
23     l_b_work = b_work.local()[:bs,:]
24     l_c = c_new.local()
25     if l_c.size > 0:
26       l_c += np.dot(l_a_work, l_b_work)
```

Figure 7.32: SUMMA Matrix Multiplication

LU Factorization

LU factorization is a classical numerical linear algebra problem that decomposes a matrix as the product of a lower triangular matrix and an upper triangular matrix. The implementation is a straightforward translation of a block LU factorization written in Matlab [100]. Our implementation is simplified by skipping partial pivoting. Figure 7.33 shows the complete source code of implementation, which makes use of the SUMMA implementation (line 47) from before (Figure 7.32) and a local LU factorization (line 25) provide by the SciPy library, which in turn uses an optimized LAPACK library. Taking advantage of the distributed array extension, we express all communication patterns through Python assignments when replicating the local LU results horizontally and vertically (line 27-31).

```
import numpy as np
from scipy import linalg
import pyHPC
from itertools import izip as zip

def lu(matrix):
  SIZE = matrix.shape[0]
  BS = np.BLOCKSIZE

  (prow,pcol) = matrix.pgrid()
  A = matrix.copy()
  L = np.zeros((SIZE,SIZE),dtype=matrix.dtype,\
               dist=True)
  U = np.zeros((SIZE,SIZE),dtype=matrix.dtype,\
               dist=True)
  for k in xrange(0,SIZE,BS):
    bs = min(BS,SIZE - k) #Current block size
    kb = k / BS # k as block index

    #Compute local LU
    slice = ((kb,kb+1),(kb,kb+1))
    for a,l,u in zip(A.blocks(slice), \
                     L.blocks(slice), \
                     U.blocks(slice)):
      (p,l[:],u[:]) = linalg.lu(a)

    #Replicate local LU horizontal and vertical
    for tk in xrange(k+bs,SIZE,BS):
      tbs = min(BS,SIZE - tk)
      L[tk:tk+tbs,k:k+bs] = U[k:k+tbs,k:k+bs]
      U[k:k+bs,tk:tk+tbs] = L[k:k+bs,k:k+tbs]

    if k+bs < SIZE:
      #Compute horizontal multiplier
      slice = ((kb,kb+1),(kb+1,SIZE/BS))
      for a,u in zip(A.blocks(slice), \
                     U.blocks(slice)):
          u[:] = np.linalg.solve(u.T,a.T).T

      #Compute vertical multiplier
      slice = ((kb+1,SIZE/BS),(kb,kb+1))
      for a,l in zip(A.blocks(slice), \
                     L.blocks(slice)):
          l[:] = np.linalg.solve(l,a)

      #Apply to remaining submatrix
      A -= pyHPC.summa(L[:,:k+bs],U[:k+bs,:])

  return (L, U)
```

Figure 7.33: LU Factorization

System	Intel Infiniband Cluster	Cray XE-6
Processor	Intel Xeon X5530	AMD Opteron 6172
Clock	2.4 GHz	2.1 GHz
Peak Performance per Core	10.6 Gflops	8.4 Gflops
Cores per NUMA Domain	4	6
NUMA Domains per Node	2	4 (packaged in 2 sockets)
Total Cores per Node	8	24
Private L1 Data Cache	64 KB	64 KB
Private L2 Data Cache	512 KB	512 KB
Shared L3 Cache per Socket	8MB	12MB
Memory Bandwidth	25.6 GB/s	25.6 GB/s
Memory per Node	24GB DDR3-1066 ECC	32GB DDR3-1066 ECC
Compiler	Intel C/C++ 11.1	PGI 11.3
Math Library	Intel MKL 10.2	Cray Scientific Library 10.5
Interconnect	Infiniband 4X QDR	Gemini 3-D Torus
Peak Bandwidth (per direction)	5 GB/s	7 GB/s
MPI	OpenMPI 1.4.2	Cray MPI 5.1.4

Table 7.4: Two distributed-memory multi-core NUMA systems for the experiments

Heat Equation

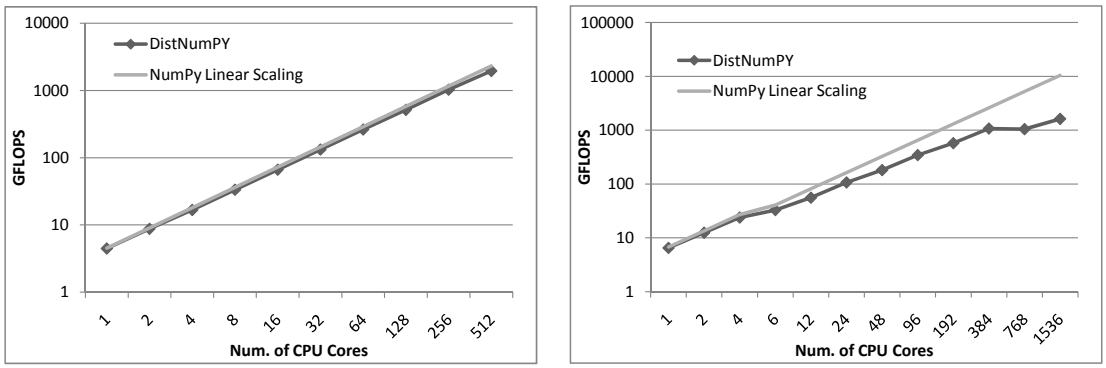
The heat equation benchmark is to solve a partial differential equation that describes the distribution of heat in a given region over time. We use the Jacobi iterative method to approximate the result with a 5-point stencil implementation. Figure 7.21 shows the computation loop of the implementation, which is concisely expressed by array operations and assignments.

7.5.5 Performance

We evaluate the performance of our benchmarks on two representative multi-core distributed memory systems – an Intel Nehalem cluster with Infiniband interconnects and a Cray XE-6 supercomputer – up to 1536 cores (Table 7.4).

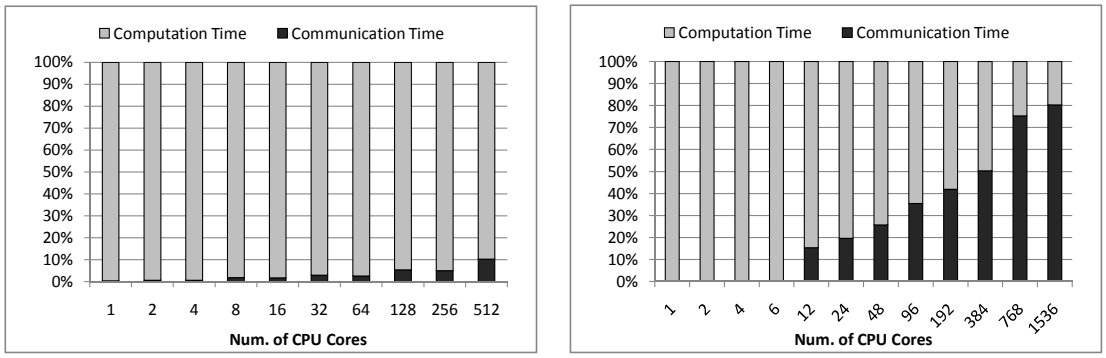
Both systems consist of multi-core Non-Uniform Memory Access (NUMA) shared-memory nodes and each node has multiple NUMA domains. CPU cores within the same NUMA domain have uniform data access latency to the local memory while CPU cores of different NUMA domains would have non-uniform data access latencies. We use hybrid parallelism, processes with threads, for all of our benchmark runs. Specifically, we run one process per NUMA domain and one thread per core within the NUMA domain. Threads within a NUMA domain communicate through shared memory and processes across NUMA domains communicate through MPI.

Through empirical study, we find that this is the configuration that achieved best performance. The two extremes usually do not work well: 1) running one process per core without threading causes too much overheads in terms of both memory footprint and communication time; 2) running one process per node and using threads across NUMA domains would also slow down the execution due to the NUMA issues with data locality and resource contention with too many threads.



(a) Intel Infiniband Cluster (b) Cray XE-6

Figure 7.34: Matrix Multiplication (SUMMA) benchmark performance



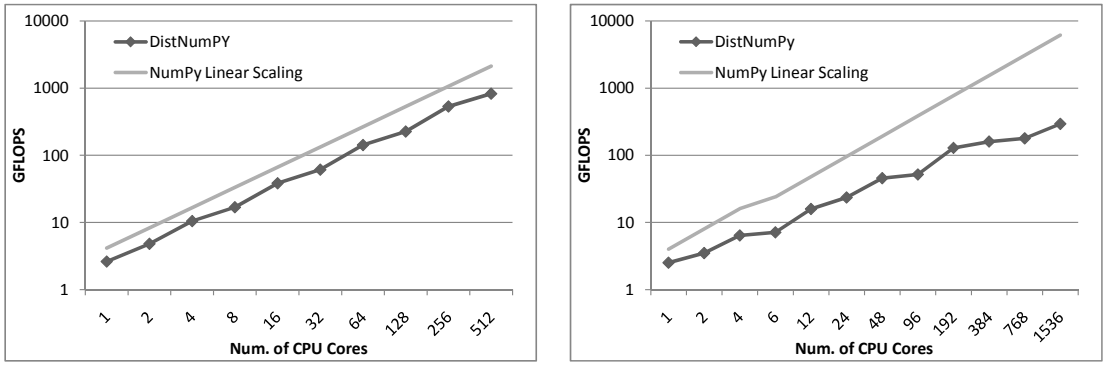
(a) Intel Infiniband cluster (b) Cray XE-6

Figure 7.35: Matrix Multiplication (SUMMA) benchmark communication and computation ratio

For each benchmark, we calculate the FLOPS based on the floating operation counts of the ideal sequential algorithm and the measured execution times. Additionally, we compare the results with the linearly scaling performance, which we calculate by extrapolating the sequential FLOPS performance of NumPy. We use this comparison as an upper bound of the achievable scalable performance. We perform weak scaling experiments, in which the problem size is scaled with the number of CPU-cores in the executions.

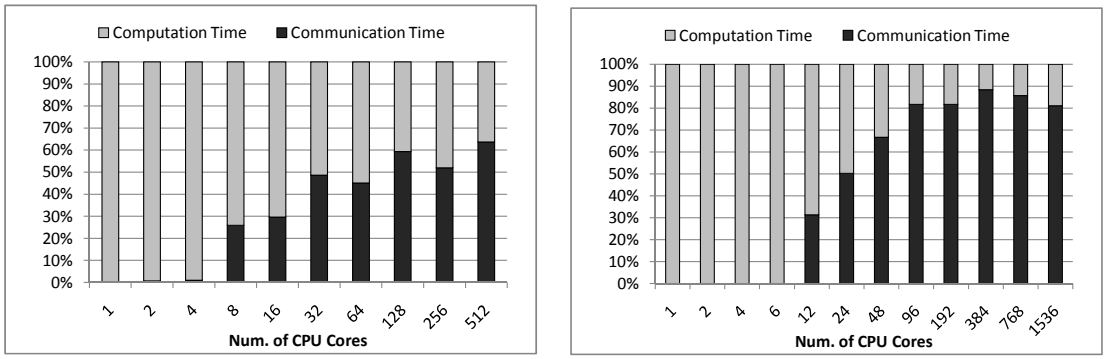
Matrix Multiplication (SUMMA)

Matrix multiplication has very high computation to communication ratio: its asymptotic computation complexity is $O(n^3)$ while its data communication complexity is only $O(n^2)$, as verified in the communication and computation ratio Figure 7.35. The SUMMA algorithm is a well-known scalable parallel algorithm for matrix multiplication. Thus this benchmark scales nearly linearly on the Intel Infiniband cluster and also performs



(a) Intel Infiniband cluster (b) Cray XE-6

Figure 7.36: LU Factorization benchmark performance



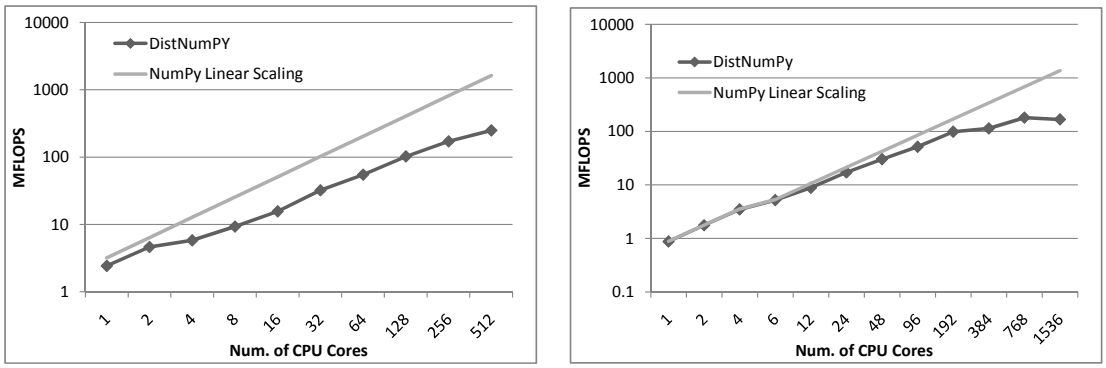
(a) Intel Infiniband Cluster (b) Cray XE-6

Figure 7.37: LU Factorization communication and computation ratio

quite well on the Cray XE-6 system as in Figure 7.34. The communication cost of running the benchmark on CPU cores in a single NUMA domain is zero because we use threads to share data directly as described before.

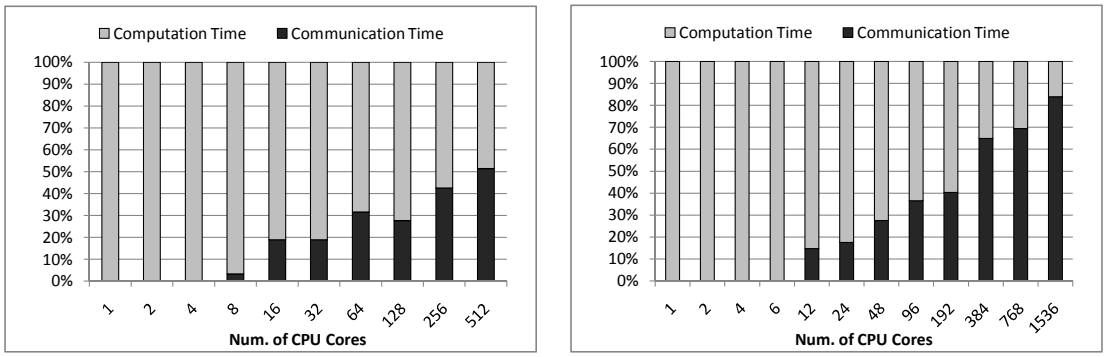
Because all local matrix computations are done with the vendor-optimized BLAS libraries (Intel MKL and Cray Scientific Library), our implementation obtains very good absolute performance in terms of the hardware peak FLOP rate on both platforms. The performance of our Python sequential implementation is very close to that of the C sequential implementation because in both cases the majority of the running time is spent in external optimized BLAS library.

The gap between our implementation "DistNumPy" and the ideal linear speed-ups of the sequential implementation is basically the overheads of performing parallel execution with the high-level abstractions in our programming model. For the Intel Infiniband cluster, the python interpreter overheads are negligible as our obtained performance tightly tracks the linear speed-up curve in Figure 7.34 (a).



(a) Intel Infiniband Cluster (b) Cray XE-6

Figure 7.38: Heat Equation benchmark performance



(a) Intel Infiniband Cluster (b) Cray XE-6

Figure 7.39: Heat Equation benchmark communication and computation ratio

LU Factorization

The LU factorization benchmark also has relatively high computation to communication ratio but does require more communication and has more data dependencies than the matrix multiplication benchmark. Thus it is expected to scales well. Figure 7.36 shows that the LU benchmark scale well on the Intel Infiniband cluster up to 512 cores. But the scalability of the LU benchmark decreases on the Cray XE-6 system when using more than 384 cores when the communication times become dominant, as shown in Figure 7.37. The Python sequential implementation of LU is about 12% and 50% slower than the C counterpart on the Intel Infiniband cluster and the Cray XE-6 system respectively. The Python implementation is slower because it performs extra work to allocate buffers and format data in addition to calling the LAPACK factorization routine.

Overall, the benchmark performance is better on the Intel Infiniband cluster than on the Cray XE-6 system as the current Cray MPI for the Cray Gemini network has limited overlapping support for non-blocking MPI communication. In addition, the job scheduler on the Cray system may allocate distant nodes to a job and the torus network

performance would suffer from the communication traffics caused by other jobs.

Heat Equation

The computation to communication ratio of this stencil-type benchmark is inherently low, which is a small constant. Thus its performance is somewhat limited by the memory bandwidth when running within a node with shared-memory and limited by the network latency and bandwidth when running on multiple nodes. Figure 7.38 shows that our implementation scales well on up to 256 cores on the Intel Infiniband cluster and up to 192 cores on the Cray XE-6 system. As the number of cores goes up, the performance is increasingly dominated by the communication time which results in suboptimal scalability (Figure 7.39). The C implementation of the Heat Equation benchmark performs much better than the Python implementation and we plan to address this performance discrepancy issue due to Python interpreter overheads in future work.

Scalability Limitations

The global address space and the SIMD execution model introduce some scalability limitations. It is nice to express data movement using the global address space but it forces each process to iterate over all elements in a global array operation not only the elements it has to compute. Thus, introducing a Python overhead that increases proportional with the global size of an array operation. In a traditional SPMD execution model the overhead is proportional with the local size of an array operation.

Another important limitation is the lack of communication latency-hiding. DistNumPy will normally use lazy evaluation to overlap communication with computation. However, the amount of instructions lazy evaluated decreases when applications use the `local` operation.

7.5.6 Conclusion

The single execution flow with fully synchronous operations of SIMD is both the main strength and weakness of data-parallel programming models: two most notorious types of parallel programming bugs, data races and deadlocks, simply do not exist in data-parallel programs because there is only one execution thread. However, it is inconvenient to perform task parallelism and conditional computations with pure data parallelism. To get the benefits of both data-parallelism and task-parallelism, we incorporate both SIMD and SPMD execution models in our programming system.

Python and other scripting languages are commonly considered as unsuitable for large scale high performance parallel computing due to its interpreter execution nature. Our work is a proof of concept that shows that Python with proper extensions and optimizations can indeed be used to develop parallel applications that scale on large distributed memory systems. The loss of raw performance due to Python interpreter overheads is considerably small because the major part of execution time is spent in the underlying computation and communication libraries. In addition, significant speedups can often be

achieved by using better algorithms and refined models, which are made much easier to implement due to the high-level abstractions of our Python-based programming system. Like Python, our programming model is general and applicable beyond scientific computing applications. A distributed shared array in our system is essentially a partitioned global address space in that the array elements may be used to store arbitrary objects.

As the ExaScale Software Study [23] pointed out: "current software approaches will be inadequate in enabling future Grand Challenge applications on Extreme Scale systems". Traditional parallel programming languages, such as C and Fortran, are good for getting hardware performance but less suitable for high-level application development and algorithm exploration. We propose a two layer approach to address the programming challenges for extreme scale systems: a low-level language layer that provides portable performance across different hardware platforms and a high-level language layer that provides portable productivity to end users. The work presented is a step towards creating such a high-level programming system and has demonstrated the feasibility of achieving productivity without compromising performance.

7.6 Summary

Overall, the DistNumPy project demonstrates that it is possible to introduce implicit data parallelism in NumPy seamlessly. Even though the parallel performance of DistNumPy is not equal manually parallelized applications, it provides good scalable performance without sacrificing the high-productivity inherited from Python/NumPy.

Chapter 8

cphVB

While developing DistNumPy I started realizing that there is no reason to only support Python/NumPy and distributed memory architectures. I strongly believe in the high-productivity and high-performance idea in DistNumPy thus I find a generalization of DistNumPy that support a broad range of languages and hardware architectures very interesting.

The project Enjing[12, 13] – a sister project of DistNumPy – demonstrates that it is possible to utilize GPGPUs[1] seamlessly in Python/NumPy applications. Encourage by this, we began the development of a new project, cphVB, that aims to provide a high-productivity and high-performance framework that supports a broad range of both programming languages and hardware architectures.

We introduce a new abstract machine framework, cphVB, that enables a vector oriented high-level programming languages to map onto a broad range of architectures efficiently. The idea is to close the gap between high-level languages and hardware optimized low-level implementations. By translating high-level vector operations into an intimidate vector byte code, cphVB enables specialized vector engines to efficiently execute the vector operations.

8.1 Introduction

Obtaining high performance from today's computing environments requires both a deep and broad working knowledge on computer architecture, communication paradigms and programming interfaces. Today's computing environments are highly heterogeneous consisting of a mixture of CPUs, GPUs, FPGAs and DSPs orchestrated in a wealth of architectures and lastly connected in numerous ways.

Utilizing this broad range of architectures manually requires programming specialists and is a very time-consuming task – time and specialization a scientific researcher typically does not have. A high-productivity language that allows rapid prototyping and still enables utilizing of a broad range of architectures is clearly preferable.

[1] General Purpose Graphics Processing Unit

There exist high-productivity language and libraries that automatically utilize parallel architectures [69][35][77]. They are however still few in numbers and have one problem in common. They are closely coupled to both the front-end, i.e. programming language and IDE, and the back-end, i.e. computing device, which makes them interesting only to the few using the exact combination of front-, and back-end.

A tight coupling between front-end technology and back-end present another problem; the usefullnes of the developed program expires as soon as the back-end does. With the rapid development of hardware architectures the time spend on implementing optimized programs for a specific hardware target is lost as soon as the hardware product expires.

In this work, we present a novel approach to the problem of closing the gap between high-productivity languages and parallel architectures, which allows a high degree of modularity and reusability. The approach involves creating a framework, cphVB, in which the computing devices (hardware) are viewed as engines that processes vectorized instructions, called Vector Engines. It defines a clear and easy to understand byte code language that the Vector Engines executes. cphVB also contains a protocol to govern the safe, and efficient exchange, creation, and destruction of model data.

cphVB provides a retargetable framework in which the user can write programs utilizing whichever cphVB supported programming interface they prefer and run the program on their own workstation while doing prototyping, such as testing correctness and functionality of their programs. Users can then deploy the exact same program in a more powerful execution environment without changing a single line of code and thus effectively solve greater problem sets.

8.1.1 Related Work

The key motivation for cphVB is to provide a framework for the utilization of heterogeneous computing systems with the goal of obtaining high performance and high productivity. Systems such as pyOpenCL/pyCUDA[66] provides a direct mapping from frontend language to the optimization target. In this case, providing the user with direct access to the low-level systems OpenCL[83] and CUDA[81] from the high-level language Python[99]. The work in [66] enables the user with the ability to write a low-level implementation combined with a high-productivity language. The goal is similar to cphVB – the approach however is entirely different. cphVB provides a means to hide low-level target specific code behind a programming model and providing a framework and runtime environment to support it.

Intel Math Kernel Library[58] is in this regard more comparable to cphVB. Intel MKL is a programming library providing utilization of multiple targets ranging from a single core CPU to a multi-core shared memory CPU and even to a cluster of computers all through the same programming API. However, cphVB is not only a programming library it is a runtime system providing support for a vector oriented programming model. The programming model is well-known from high-productivity languages such as MATLAB [74], R[96], IDL[92], GNU Octave[41] and Numerical Python (NumPy) [82] to name a few.

cphVB is more closely related to the work described in [46], here a compilation framework is provided for execution in a hybrid environment consisting of both CPUs and GPUs. Their framework uses a Python/NumPy based frontend that uses Python decorators as hints to do selective optimizations. cphVB similarly provides a NumPy based frontend and equivalently does selective optimizations. However, cphVB uses a slightly less obtrusive approach; program selection hints are sent from the frontend via the NumPy-bridge. This approach provides the advantage that any existing NumPy program can run unaltered and take advantage of cphVB without changing a single line of code. Whereas unPython requires the user to manually modify the source code by applying hints in a manner similar to that of OpenMP[84]. This non-obtrusive design at the source level is to the author's knowledge novel.

Microsoft Accelerator[35] introduces ParallelArray, which is similar to the utilization of the NumPy arrays in cphVB but there are strict limitations to the utilization of ParallelArrays. ParallelArrays does not allow the use of direct indexing, which means that the user must copy a ParallelArray into a conventional array before indexing. cphVB instead allows indexed operations and additionally supports array-views, which are array-aliases that provide multiple ways to access the same memory allocation. Thus, the data structure in cphVB is highly flexible and provides elegant programming solutions for a broad range of numerical algorithms. Intel provides a similar approach called Intel Array Building Blocks (ArBB) [77] that provides retargetability and dynamic compilation. It is thereby possible to utilize heterogeneous architectures from within standard C++. The retargetability aspect of Intel ArBB is represented in cphVB as a plain and simple configuration file that define the cphVB runtime environment. Intel ArBB provides a high performance library that utilizes a heterogeneous environment and hides the low-level details behind a vector oriented programming model similar to cphVB. However, ArBB only provides access to the programming model via C++ whereas cphVB is not biased towards any one specific frontend language.

On multiple points cphVB is closely related in functionality and goals to the SEJITS [27] project. SEJITS takes a different approach towards the frontend and programming model. SEJITS provides a rich set of computational kernels in a high-productivity language such as Python or Ruby. These kernels are then specialized towards optimality criteria. This approach has shown to provide performance that at times out-performs even hand-written specialized code towards a given architecture. Being able to construct computational kernels is a core issue in data-parallel programming.

The programming model in cphVB does not provide this kernel methodology. cphVB has a strong NumPy heritage which also shows in the programming model. The advantage is easy adaptability of the cphVB programming model for users of NumPy, Matlab, Octave and R. The cphVB programming model is not a stranger to computational kernels – cphVB deduce computational kernels at runtime by inspecting the vector bytecode generated by the Bridge.

cphVB provides in this sense a virtual machine optimized for execution of vector operations, previous work [6] was based on a complete virtual machine for generic execution whereas cphVB provides an optimized subset.

8.2 Target Programming Model

To hide the complexities of obtaining high-performance from a heterogeneous environment any given system must provide a meaningful high-level abstraction. This can be realized in the form of domain specific languages, embedded languages, language extensions, libraries, APIs etc. Such an abstraction serves two purposes: 1) It must provide meaning for the end-user such that the goal of high-productivity can be met with satisfaction. 2) It must provide an abstraction that consists of a sufficient amount of information for the system to optimize its utilization.

cphVB is not biased towards any specific choice of abstraction or frontend technology as long as it is compatible with a vector oriented programming model. This provides means to use cphVB in functional programming languages, provide a frontend with a strict mathematic notation such as APL[51] or a more relaxed syntax such as MATLAB.

8.3 Design of cphVB

The key contribution in cphVB is a framework that support a vector oriented programming model. The idea of cphVB is to provide the mechanics to seamlessly couple a programming language or library with an architecture-specific implementation of vectorized operations.

cphVB consists of a number of components that communicate using a simple protocol. Components are allowed to be architecture-specific but they are all interchangeable since all uses the same communication protocol. The idea is to make it possible to combine components in a setup that perfectly match a specific execution environment. cphVB consist of the following components:

- **Programming Interface** The programming language or library exposed to the user. cphVB was initially meant as a computational back-end for the Python library NumPy, but we have generalized cphVB to potential support all kind of languages and libraries. Still, cphVB has design decisions that are influenced by NumPy and its vector objects.
- **Bridge** The role of the Bridge is to introduce cphVB into an already existing languages and libraries. The Bridge generates the cphVB byte code that corresponds to the user-code.
- **Vector Engine** The Vector Engine is the architecture-specific implementation that executes cphVB byte code.
- **Vector Engine Manager** The Vector Engine Manager manages data location and ownership of vectors. It also manages the distribution of computing jobs between potentially several Vector Engines, hence the name.

An overview of the design can be seen in Figure 8.1.

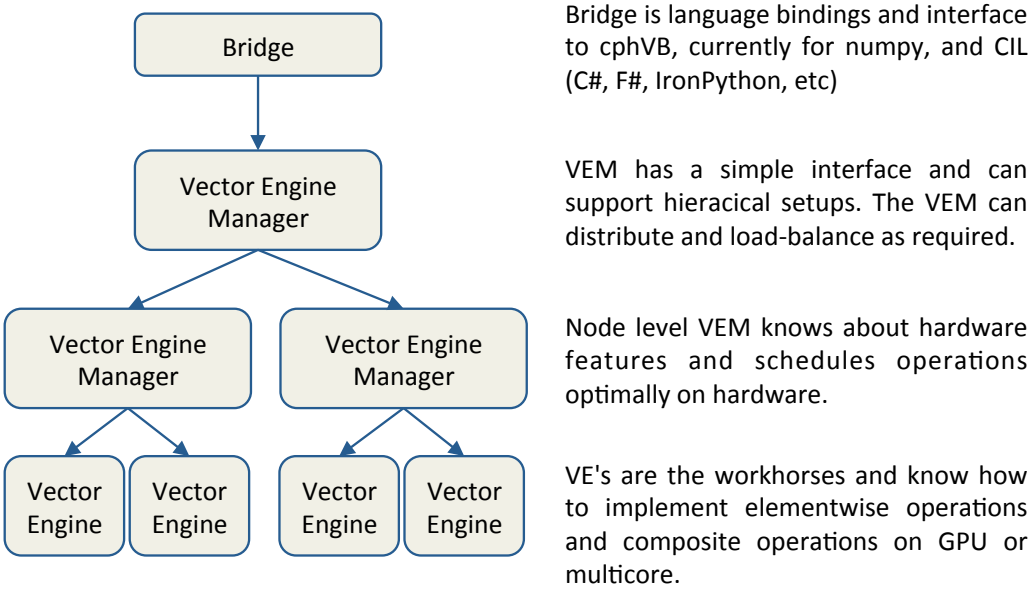


Figure 8.1: Design Overview of cphVB

```
#Root of the setup
[setup]
bridge = numpy
debug = true

#Bridge for NumPy
[numpy]
type = bridge
children = node

#Vector Engine Manager for a single machine
[node]
type = vem
impl = libcphvb_vem_node.so
children = pthread

#Vector Engine implemented using Posix Threads
[pthread]
type = ve
impl = libcphvb_ve_pthd.so
```

Figure 8.2: Configuration ini-file

8.3.1 Configuration

To make cphVB as flexible a framework as possible, we manage the setup of all the components at runtime through a configuration file. The idea is that the user can change the setup of components simply by editing the configuration file before executing the user application. Additionally, the user only has to change the configuration file in order to run the application on different systems with different computational resources. The configuration file uses the ini syntax – Figure 8.2 is an example of a setup for NumPy executing parallel on one machine using Pthreads.

8.3.2 Byte Code

The central part of the communication between all the components in cphVB is vector byte code. The goal with the byte code language is to be able to express operations on multidimensional vectors. Taking inspiration from single instruction, multiple data (SIMD) instructions but adding structure to the data. This, of course, fits very well with the array operations in NumPy but is not bound to nor limited to these.

We would like the byte code to be a concept that is easy to explain and understand. It should have a simple design that is easy to implement. It should be easy and inexpensive to generate and decode. To fulfill these goals we chose a design that conceptually is an assembly language where the operands are multidimensional vectors. Furthermore, to simplify the design the assembly language should have a one-to-one mapping between instruction mnemonics and opcodes.

In the basic form, the byte-code instructions are primitive operations on data, e.g.

addition, subtraction, multiplication, division, square root etc. As an example, let us look at addition. Conceptually it has the form:

```
add $d, $a, $b
```

Where `add` is the opcode for addition. After execution `$d` will contain the sum of `$a` and `$b`.

The requirement is straightforward: we need an opcode. The opcode will explicitly identify the operation to perform. Additionally the opcode will implicitly define the number of operands. Finally, we need some sort of symbolic identifiers for the operands. Keep in mind that the operands will be multidimensional arrays.

8.3.3 Interface

The Vector Engine and the Vector Engine Manager exposes simple API that consists of the following functions: initialization, finalization, registration of a user-defined operation and execution of a list of byte codes. Furthermore, the Vector Engine Manager exposes a function to define new arrays.

8.3.4 Bridge

The Bridge is the *bridge* between the programming interface, e.g. Python/NumPy, and the Vector Engine Manager. The Bridge is the only component that is specifically implemented for the programming interface. In order to add cphVB support to a new language or library, one only has to implement the bridge component. It generates byte code based on programming interface and sends them to the Vector Engine Manager.

8.3.5 Vector Engine Manager

Instead of just letting the front-end communicate directly with the Vector Engine, we introduced a Vector Engine Manager (VEM) into the design. It is the responsibility of the VEM to manage data ownership and distribute byte code instructions to several Vector Engines.

For efficiency reasons, the VEM handles instantiating and discarding arrays. If the programming interface or the Bridge controls this, they would always have to copy data from main memory to the device that is going to do the calculations. Often arrays are created with structured data (e.g. random, constants), with no data at all (e.g. empty), or as a result of calculation. In any case it saves, potentially several, memory copies to delay the actual memory allocation. Typically, array data will exist on the computing device exclusively.

In order to minimize data copying we introduce a data ownership scheme. It keeps track of which components in cphVB that needs to access a given array. The goal is to allow the system to have several copies of the same data while ensuring that they are in synchronization. We base the data ownership scheme on three instructions, *sync*, *release* and *discard*:

- **Sync** is used to request read access to a data object. This means that when acknowledging a *sync* request, the copy existing in shared memory needs to be the most resent copy.
- **Discard** is used to signal that the copy in shared memory has been updated and all other copies are now invalid. Normally used for upgrading a read access to a write access.
- **Release** is simply the same as a *sync* followed by a *discard*. This is used for requesting write access.

The cphVB components follow the following four rules when implementing the data ownership scheme:

1. The Bridge will always ask the Vector Engine Manager for access to a given data object. It will send a *sync* request for read access and a *release* request for write access. The Bridge will not keep track of ownership itself.
2. A Vector Engine can assume that it has write access to all of the output parameters that are referenced in the instructions it receives. Likewise, it can assume read access on all input parameters.
3. A Vector Engine is free to manage its own copies of arrays and implement its own scheme to minimize data copying. It just needs to copy modified data back to share memory when receiving a *sync* instruction and delete all local copies when receiving a *discard* instruction. A *release* instruction can be handled as a*sync* followed by a *discard* instruction.
4. The Vector Engine Manager keeps track of array ownership for all its children. The owner of an array has full (i.e. write) access. When the parent component of the Vector Engine Manager, normally the Bridge, request access to an array, the Vector Engine Manager will forward the request to the relevant child component. The Vector Engine Manager never accesses the array itself.

The Vector Engine Manager also keeps track of how many references there is to any given array. If there are no more references to an array it deallocates memory and sends discard instructions to any child component that may have a local copy.

Additionally, the Vector Engine Manager needs the capability to handle multiple children components. In order to maximize parallelism the Vector Engine Manager can distribute workload and array data between its children components.

8.3.6 Vector Engine

Though the Vector Engine is the most complex component of cphVB, it has a very simple and a clearly defined role. It has to execute all instructions it receives in a manner that obey the serialization dependencies between instructions. Finally, it has to ensure that the rest of the system has access to the results as governed by the rules of the *sync*, *release*, and *discard* instructions.

8.4 Implementation of cphVB

In order to demonstrate our cphVB design we have implemented a basic cphVB setup. This concretization of cphVB is by no means exhaustive. The setup is targeting the NumPy library executing on a single machine with multiple CPU-cores. In this section, we will describe the implementation of each component in the cphVB setup – the Bridge, the Vector Engine Manager, and the Vector Engine. The cphVB design rules (Section 8.3) govern the interplay between the components.

8.4.1 Bridge

The role of the Bridge is to introduce cphVB into an already existing project. In this specific case NumPy, but could just as well be "R" or any other language/tool that works primarily on vectorizable operations on large data objects.

It is the responsibility of the Bridge to generate cphVB instructions on basis of the Python program that is being run. The NumPy Bridge is an extension of NumPy version 1.6. It uses hooks to divert function call where the program access cphVB enabled NumPy arrays. The hooks will translate a given function into its corresponding cphVB byte code when possible. When it is not possible, the hooks will feed the function call back into NumPy and thereby forcing NumPy to handle the function call itself.

The Bridge operates with two address spaces for arrays: the cphVB space and the NumPy space. All arrays starts in the NumPy space as a default. The original NumPy implementation handles these arrays and all operations using them. It is possible to assign an array to the cphVB space explicitly by using an optional cphVB parameter in array creation functions such as `empty` and `random`. The cphVB bridge implementation handles these arrays and all operations using them.

In two circumstances, it is possible for an array to transfer from one address space to the other implicitly at runtime.

1. When an operation accesses an array in the cphVB address space but it is not possible for the bridge to translate the operation into cphVB code. In this case, the bridge will synchronize and move the data to the NumPy address space. For efficiency no data is actually copied instead the bridge uses the `mremap`[2] function to re-map the relevant memory pages.

2. When an operations access arrays in different address spaces the Bridge will transfer the arrays in the NumPy space to the cphVB space. Afterwards, the bridge will translate the operation into byte code that cphVB can execute.

In order to detect direct access to arrays in the cphVB address space by the user, the original NumPy implementation, a Python library or any other external source, the bridge protects the memory of arrays that are in the cphVB address space using `mprotect`[3]. Because of this memory protection, subsequently accesses to the memory

[2]The function `mremap()` in GNU C library 2.4 and greater.
[3]The function `mprotect()` in the POSIX.1-2001 standard.

will trigger a segmentation fault. The Bridge can then handle this kernel signal by transferring the array to the NumPy address space and cancel the segmentation fault. This technique makes it possible for the Bridge to support all valid Python/NumPy application since it can always fallback to the original NumPy implementation.

In order to gather greatest possible information at runtime, the Bridge will collect a batch of instructions rather than executing one instruction at a time. The Bridge will keep recording instruction until either the application reaches the end of the program or untranslatable NumPy operations forces the Bridge to move an array to the NumPy address space. When this happens, the Bridge will call the Vector Engine Manager to execute all instructions recorded in the batch.

8.4.2 Vector Engine Manager

The Vector Engine Manager (VEM) in our setup is very simple because it only has to handle one Vector Engine thus all operations go to the same Vector Engine. Still, the VEM creates and deletes arrays based on specification from the Bridge and handles all meta-data associated with arrays.

8.4.3 Vector Engine

In order to maximize the CPU cache utilization and enables parallel execution the first stage in the VE is to form a set of instructions that enables data blocking. That is, a set of instructions where all instructions can be applied on one data block completely at a time without violating data dependencies. This set of instructions will be referred to as a kernel.

The VE will form the kernel based on the batch of instructions it receives from the VEM. The VE examines each instruction sequentially and keep adding instruction to the kernel until it reaches an instruction that is not *blockable* with the rest of the kernel. In order to be blockable with the rest of the kernel an instruction must satisfy the following two properties where A is all instructions in the kernel and N is the new instruction.

1. The input arrays of N and the output array of A do not share any data or represents precisely the same data.

2. The output array of N and the input and output arrays of A do not share any data or represents precisely the same data.

When the VE has formed a kernel, it is ready for execution. Since all instruction in a kernel supports data blocking the VE can simply assign one block of data to each CPU-core in the system and thus utilizing multiple CPU-cores. In order to maximize the CPU cache utilization the VE may divide the instructions into even more data blocks. The idea is to access data in chunks that fits in the CPU cache. The user, through an environment variable, manually configures the number of data blocks the VE will use.

Processor	Two Intel Xenon
Clock	2.67 GHz GHz
Private L1 Data Cache	64 KB
Private L2 Data Cache	512 KB
Shared L3 Cache per Socket	12MB
Memory Bandwidth	25.6 GB/s
Memory per Node	96GB DDR3-1066
Compiler	GCC 4.4.5

Table 8.1: Lenovo ThinkStation D20

8.5 Performance Study

In order to demonstrate the performance of our initial cphVB implementation and thereby the potential of the cphVB design, we will conduct some performance benchmarks using NumPy[4]. We execute the benchmark applications on one Lenovo ThinkStation D20 with two Intel Xenon processers (Table 8.1). The experiments use up to all eight CPU-cores on the machine and for each executaion we calculate the speedup of cphVB compared to NumPy. We perform strong scaling experiments, in which the problem size is constant though all the executions. For each experiment, we find the block size that results in best performance and we calculate the result of each experiment using the average of three executions.

The benchmark consists of the following Python/NumPy applications. All are pure Python applications that make use of NumPy and none uses any external libraries.

- **Monte Carlo Pi** Approximating Pi using Monte Carlo simulation. The implementation is a translation and vectorization of the Monte Carlo simulation included in the benchmark suite SciMark 2.0[86], which is written in Java (Figure 8.3).
- **kNN** A naïve implementation of a k Nearest Neighbor search (Figure 8.4).
- **N-body** A Newtonian N-body simulation that uses a $O(n^2)$ algorithm that computes all body-body interactions. (Figure 8.5).
- **Shallow Water** A simulation that simulates a system governed by the shallow water equations. It is a translation of a MATLAB application by Burkardt[21] (Figure 8.6).

8.5.1 Discussion

The Monte Carlo Pi simulation is an embarrassingly parallel problem because thread coordination is only relevant at the end of the program. Thus, cphVB provides good performance speedup compared to NumPy – at eight CPU-cores cphVB is more than six times faster than NumPy.

[4]NumPy version 1.6.1

On the other hand, our naïve implementation of the k Nearest Neighbor search is not an embarrassedly parallel problem. However, it has a time complexity of $O(n^2)$ when the number of elements and the size of the query set is n, thus the problem should be scalable. The result of our experiment is also promising – with a performance speedup of more than five when running on eight CPU-cores. Because of better cache utilization through data blocking, the performance of cphVB is more than twice as good as NumPy when using one CPU-core. Still, when using more than four CPU-cores the memory bandwidth becomes the limiting factor.

The N-body simulation also has a time complexity of $O(n^2)$ but it exhibits less cache locality. At one CPU-core NumPy outperforms cphVB by quite a margin. This is because the N-body implementation uses matrix transpose as part of the computation loop, which NumPy controls more efficiently than cphVB.

Finally, the Shallow Water simulation only has a time complexity of $O(n)$ thus it is the most memory intensive application in our benchmark. Still, cphVB manages to achieve a performance speedup of almost three compared to NumPy.

8.6 Summary

The vector oriented programming model used in cphVB provides a framework for high-performance and high-productivity. It enables the end-user to execute vectorized applications on a broad range of hardware architectures efficiently without any hardware specific knowledge. Furthermore, the cphVB design supports scalable architectures such as clusters and supercomputers. It is even possible to combine architectures in order to exploit hybrid programming where multiple levels of parallelism exist. The authors in [68] demonstrate that combining shared memory and distributed memory parallelism through hybrid programming is essential in order to utilize the Blue Gene/P architecture fully.

In a case study, we demonstrate the design of cphVB by implementing a frontend for Python/NumPy that targets multi-core CPUs in a shared memory environment. The implementation executes vectorized applications in parallel without any user intervention. Thus showing that it is possible to retain the high abstraction level of Python/NumPy while fully utilizing the underlying hardware. Furthermore, the implementation demonstrates scalable performance – a k-nearest neighbor search purely written in Python/NumPy obtains a speedup of more than five compared to a native execution.

Future work will further test the cphVB design model as new frontend technologies and heterogeneous architectures are supported.

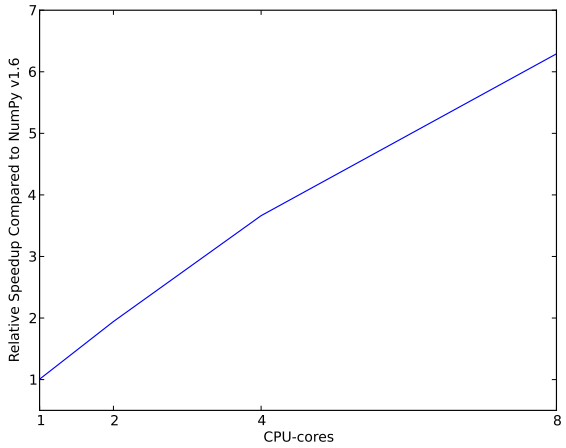


Figure 8.3: Performance of the Monte Carlo Pi Approximation. The job consists of a vector with 100M elements using 10 iterations.

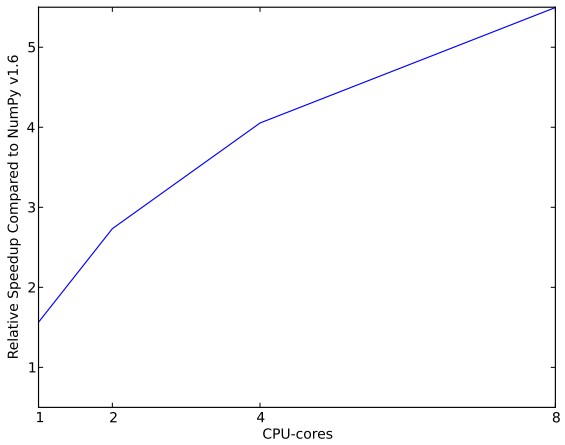


Figure 8.4: Performance of the k Nearest Neighbor search. The job consists of 1K elements and the query set also consists of 1K elements.

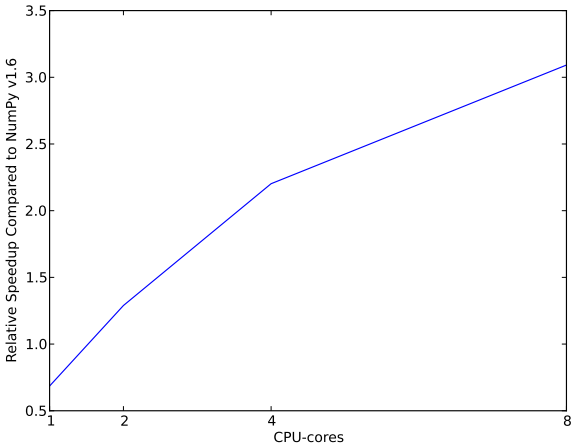


Figure 8.5: Performance of the N-body simulation. The job consists of 8K bodies that simulate 10 time steps.

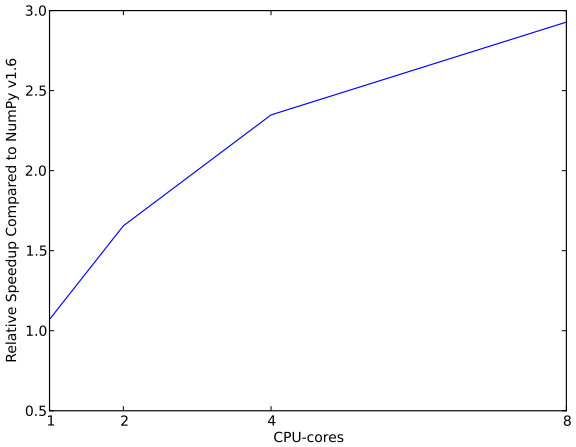


Figure 8.6: Performance of the Shallow Water Equation. The job consists of 4M grid points that simulate 10 time steps.

Chapter 9

Future Work

The development of the three projects discussed in this thesis, GPAW, DistNumPy and cphVB, is not finished. Of the three projects, GPAW is the most mature project but still work remains if the entire GPAW computation should utilize latency-hiding and hybrid programming.

DistNumPy

The DistNumPy project is still in the early development stage. We have introduced communication latency-hiding and full array-view support but the result from our benchmarks shows that memory latency is another aspect that is important to address. One approach to address this issue is to extend our latency-hiding model with cache locality optimization. The scheduler will have to prioritize array operations that are likely to be in the cache. Additionally, *just in time* compilation can reduce the DistNumPy overhead and make array operations more CPU-intensive and by merging array operations together into a joint operation. Finally, the introduction of hybrid programming could improve the utilization of shared memory nodes.

One of the main features in DistNumPy when using the SIMD execution model is automatic communication latency hiding. This feature is disabled somewhat when mixing SIMD and SPMD. The problem is that DistNumPy has to execute all lazy evaluated operations when the user changes to the SPMD model, e.g. when the user calls `local` in order to apply computations based on process affinity. Therefore, DistNumPy will not automatically overlap local operations with communication.

The introduction of a *scheduler* function in DistNumPy could solve the problem. The user would then use this new function to schedule local operations instead of applying them immediately. This would enable DistNumPy to include the local operation in its lazy evaluation system thus making it possible to overlap local operations with communication.

One the other hand, it is harder to address the scalability limitation introduced by using the global address space. The use of global data iterations is very natural when programming using the global address space. Still, it is possible to limit this issue by

introducing dedicate global functions, such as functions to move or replicate data.

cphVB

The cphVB project is also in the early development stage. The future goals of cphVB involve improvement in two major areas; expanding support and improving performance. Work has started on a CIL-bridge, which will leverage the use of cphVB to every CIL based programming language, which among others include: C#, Visual C++ and VB.NET. Another project in current progress within the area of support is a C++ bridge providing a library-like interface to cphVB using overloading and templates to provide a high-level interface in C++.

To improve both support and performance, work is in progress on a vector engine targeting OpenCL compatible hardware, mainly focusing on using GPU-resources to improve performance. Additionally, we plan to convert DistNumPy into a vector engine manager that targets distributed memory architectures and thereby merging the two projects.

In terms of pure performance enhancement, cphVB will introduce JIT compilation in order to improve memory intensive applications. The current vector engine for multi-cores CPUs uses data blocking to improve cache utilization but as our experiments show then the memory intensive applications still suffer from the von Neumann bottleneck[8]. By JIT compile the instruction kernels; it is possible to improve cache utilization drastically.

Chapter 10

Conclusion

In this thesis, I explore the possibility of parallel execute sequential scientific applications seamlessly. Essential for my work is vector-oriented parallelism as a high-productivity programming approach to develop applications that targets a broad range of parallel hardware architectures. I introduce data parallelism implicitly to the numerical Python library, NumPy, where its emphasis on high-level array operations matches very well data parallelism.

I realized this high-productivity potential of NumPy through my involvement in the scientific application GPAW. In turns out that, GPAW uses Python/NumPy as the main development language but because of the extreme scalable performance requirement GPAW resort to the more low-level C language and using MPI for the parallelization. Though architecture optimizations, such as communication latency hiding and hybrid programming, I managed a CPU utilization of 70% on Blue Gene/P supercomputer using 16384 CPU-cores.

Because of a large user base, this low-level approach is practical for the GPAW project. However, in smaller projects, which do not have the resources to employ dedicated developers, this is an impracticable approach. Therefore, I introduce the project, DistNumPy, to parallelize Python/NumPy application automatically. The performance of DistNumPy is not equal manually parallelized applications but it provides good scalable performance without sacrificing the high-productivity inherited from Python/NumPy.

The DistNumPy project targets a single programming language, Python, and a single hardware architecture, cluster of single-core CPUs, exclusively. DistNumPy do support multi-core CPUs but all optimizations targets single-cores nodes in a distributed memory cluster. We strongly believe in the high-productivity and high-performance idea in DistNumPy thus we find a generalization of DistNumPy that support a broad range of languages and hardware architectures very interesting. Therefore, we introduce new abstract machine framework, cphVB, that enables a vector oriented high-level programming languages to map onto a broad range of architectures efficiently.

The vector oriented programming model used in cphVB provides a framework for high-performance and high-productivity. It enables the end-user to execute vectorized

applications on a broad range of hardware architectures efficiently without any hardware specific knowledge. Furthermore, the cphVB design supports scalable architectures such as clusters and supercomputers. It is even possible to combine architectures in order to exploit hybrid programming where multiple levels of parallelism exist.

In a case study, we demonstrate the design of cphVB by implementing a frontend for Python/NumPy that targets multi-core CPUs in a shared memory environment. The implementation executes vectorized applications in parallel without any user intervention thus showing that it is possible to retain the high abstraction level of Python/NumPy while fully utilizing the underlying hardware.

cphVB is the more promising of the two projects, DistNumPy and cphVB, because it generalize the idea in DistNumPy. In a sense, the DistNumPy project is a subset of the cphVB project because DistNumPy essential is a cphVB implementation that only supports Python/NumPy and distributed memory clusters.

Generally, the work shows that it is indeed possible to hide parallelism from the programmer without designing a new programming language (X10, Fortress, etc.). However, the amount of parallelism is limited to the use of high-level array operations.

Bibliography

[1] MPI for Python. http://mpi4py.scipy.org/.

[2] UPC Language Specifications, v1.2. Technical Report LBNL-59208, 2005.

[3] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener. Fortran 90 Handbook. *Intertext-McGraw Hill*, 1992.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques and tools*. Addison-Wesley, Reading, MA, 1986.

[5] E. Allen, D. Chase, J. Hallett, V. Luchangco, J. W. Maessen, S. Ryu, G. L. Steele Jr, S. Tobin-Hochstadt, J. Dias, and C. Eastlund. The Fortress language specification v1.0. *Sun Microsystems*, 2008.

[6] Rasmus Andersen and Brian Vinter. The Scientific Byte Code Virtual Machine. In *Proceedings of the 2008 International Conference on Grid Computing & Applications, GCA 2008 : Las Vegas, Nevada, USA, July 14-17, 2008. CSREA Press.*, pages 175–181, 2008.

[7] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: a portable linear algebra library for high-performance computers. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[8] J. Backus. Can Programming be Liberated from the von Neumann Style?: A Functional Style and its Algebra of Programs. *Communications of the ACM*, 16(8):613–641, 1978.

[9] R. Bariuso and A. Knies. SHMEM's User's Guide: Cray Research. *Inc., SN-2516, rev*, 2(2), 1994.

[10] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. Entering the petaflop era: the architecture and performance of Roadrunner. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[11] Siegfried Benkner, Piyush Mehrotra, John Van Rosendale, and Hans Zima. High-level management of communication schedules in HPF-like languages. In *Proceedings of the 12th international conference on Supercomputing*, ICS '98, pages 109–116, New York, NY, USA, 1998. Institute for Software Technology and Parallel Systems, University of Vienna.

[12] Morten A. Bentsen. Enjing A JIT Compiling CUDA Backend for NumPy. Master's thesis, University of Copenhagen, Denmark, June 2010.

[13] Morten A. Bentsen and Brian Vinter. Enjing - a JIT Backend for CUDA Devices. *Second Workshop on Programming Models for Emerging Architectures*, 2010.

[14] F. Black and M. Scholes. The pricing of options and corporate liabilities. *The journal of political economy*, pages 637–654, 1973.

[15] Laura Susan Blackford. ScaLAPACK. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing 96 Supercomputing 96*, page 5, 1996.

[16] P. E. Blochl. Projector augmented-wave method. *Phys. Rev. B*, 50(24):17953–17979, 1994.

[17] Troels Blum. Generalizing Execution of Vectorizable Computations by Generating Vector Oriented Byte Code. Master's thesis, University Of Copenhagen, Denmark, April 2011.

[18] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King K. Su. Myrinet – A Gigabit-per-Second Local-Area-Network. *j-IEEE-MICRO*, 15(1):29–36, 1995.

[19] Dan Bonachea. GASNet Specification, v1.1. Technical Report UCB/CSD-02-1207, University of California at Berkeley, Berkeley, CA, USA, 2002.

[20] T. Brandes and F. Zimmermann. ADAPTOR - A transformation tool for HPF programs. 1994.

[21] John Burkardt. Shallow Water Equations, 2010.

[22] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38, 2009.

[23] Dan Campbell, Mary Hall, William Harrod, Jon Hiller, David Koester, John Levesque, Robert Schreiber, and Allan Snavely. ExaScale Software Study : Software Challenges in Extreme Scale Systems Exascale Software Study : Software Challenges in Extreme Scale Systems. *Government PROcurement*, pages 1–159, 2009.

[24] Franck Cappello and Daniel Etiemble. MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks. *SC Conference*, 0:12, 2000.

[25] William W. Carlson, Jesse M. Draper, David Culler, Kathy Yelick, Eugene Brooks, and Karren Warren. Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, Bowie, MD, May 1999.

[26] H. Casanova. N-body Simulation Assignment, Nov 2008.

[27] Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanović, James Demmel, Kurt Keutzer, John Shalf, Kathy Yelick, and O. Fox. SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization. In *Proc of 1st Workshop Programmable Models for Emerging Architecture PMEA*, number UCB/EECS-2010-23. Citeseer, 2009.

[28] B.L. Chamberlain. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291, 2007.

[29] Bradford L. Chamberlain, Sung-Eun Choi, E. Christopher Lewis, Calvin Lin, Lawrence Snyder, and Derrick Weathersby. ZPL: A Machine Independent Programming Language for Parallel Computers. *Software Engineering*, 26(3):197–211, 2000.

[30] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA '05*, pages 519–538, New York, NY, USA, 2005. ACM.

[31] R. Choy and A. Edelman. MATLAB*P 2.0: A unified parallel MATLAB. Technical report, Massachusetts Institute of Technology, January 2003.

[32] R. Choy and A. Edelman. Parallel MATLAB: Doing it Right. *Proceedings of the IEEE*, 93(2):331, 2005.

[33] C. J. Conti. Concepts for buffer storage. *IEEE Computer Group News*, 2(8):9–13, 1969.

[34] L. Dagum and R Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46, 1998.

[35] Tarditi David, Puri Sidd, and Oglesby Jose. Accelerator : Using Data Parallelism to Program GPUs for General-Purpose Uses. *October*.

[36] M. J. Daydé and I. S. Duff. Use of parallel level 3 BLAS in LU factorization on three vector multiprocessors the ALLIANT FX/80, the CRAY-2, and the IBM 3090 VF. *ACM SIGARCH Computer Architecture News*, 18(3):82, 1990.

[37] Jack J. Dongarra. Performance of Various Computers Using Standard Linear Equations Software. Technical Report CS-89-85, University of Tennessee, dec 1990.

[38] Jack J. Dongarra and R. Clint Whaley. LAPACK Working Note 94: A User's Guide to the BLACS v1.0. Technical Report UT-CS-95-281, Department of Computer Science, University of Tennessee, mar 1995.

[39] J.J. Dongarra, J.R. Bunch, C.B. Moler, and G.W. Stewart. LINPACK users' guide. *SIAM*, 1, 1979.

[40] Paul F. Dubois. Guest Editor's Introduction: Python: Batteries Included. *Computing in Science Engineering*, 9(3):7–9, may-june 2007.

[41] John W. Eaton. GNU Octave. *History*, 103(February):1–356, 1997.

[42] High Performance Fortran Forum. High Performance Fortran Language Specification, version 1.0. Technical Report CRPC-TR92225, Houston, Tex., 1993.

[43] Nathan R. Fredrickson, Ahmad Afsahi, and Ying Qian. Performance characteristics of openMP constructs, and application benchmarks on a large symmetric multiprocessor. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 140–149, New York, NY, USA, 2003. ACM.

[44] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[45] Rahul Garg and José Nelson Amaral. Compiling Python to a hybrid execution environment. In *GPGPU '10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 19–30, New York, NY, USA, 2010. ACM.

[46] Rahul Garg and José Nelson Amaral. Compiling Python to a hybrid execution environment. *Computing*, pages 19–30, 2010.

[47] Robert A. van de Geijn and Jerrell Watts. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency - Practice and Experience*, 9(4):255–274, 1997.

[48] D. H. Gibson. Considerations in block-oriented systems design. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 75–80, New York, NY, USA, 1967. ACM.

[49] William Gropp. MPICH2: A New Start for MPI Implementations. *Lecture Notes in Computer Science*, 2474:7, 2002.

[50] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI Portable Parallel Programming with the Message Passing Interface*. The MIT Press, 1994.

[51] C. Guide, F. Love, H. Loan, K. Kreuk, P. Hilton, and K.E. Iverson. APL (programming Language). Website.

[52] M. U. Guide. The MathWorks. *Inc., Natick, MA*, 5, 1998.

[53] Iain Haslam. 3D Lattice Boltzmann (BGK) model of a fluid, 2006.

[54] D. S. Henty. Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling. *Supercomputing, ACM/IEEE 2000 Conference*, pages 10–10, 2000.

[55] Jon Hill, Matthew Hambley, Thorsten Forster, Muriel Mewissen, Terence M Sloan, Florian Scharinger, Arthur Trew, and Peter Ghazal. SPRINT: a new parallel framework for R. *BMC Bioinformatics*, 9:558, 2008.

[56] M. Hipp and W. Rosenstiel. *Parallel Hybrid Particle Simulations Using MPI and OpenMP*, pages 189–197. Springer-Verlag Berlin Heidelberg, 2004.

[57] J. Hollingsworth, K. Liu, and V. Paúl Pauca. Parallel Toolbox for MATLAB. Technical report, Winston-Salem, NC, USA, 1996.

[58] Intel. Transform, 2008.

[59] Jacob Jensen. Suitability of the CBE for a scientific program. Master's thesis, Department of computer science. University of Copenhagen, 2008.

[60] Weihang Jiang, Liuxing Liu, Hyun-Wook Jin, D.K. Panda, W. Gropp, and R. Thakur. High performance MPI-2 one-sided communication over infinilband. In *IEEE International Symposium on Cluster Computing and the Grid, 2004. CCGrid 2004.*, page 531, 2004.

[61] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.

[62] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4):589, 2005.

[63] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of High Performance Fortran: an historical object lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 7–1, New York, NY, USA, 2007. ACM.

[64] A. A. Khan, C. L. McCreary, and M. S. Jones. A Comparison of Multiprocessor Scheduling Heuristics. *Parallel Processing, International Conference on*, 2:243–250, 1994.

[65] Michael Kistler, John Gunnels, Daniel Brokenshire, and Brad Benton. Petascale computing with accelerators. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 241–250, New York, NY, USA, 2009. ACM.

[66] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Brain*, 911(4):1–24, 2009.

[67] C. Koelbel and P. Mehrotra. Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):440–451, 1991.

[68] M. Kristensen, H. Happe, and B. Vinter. Hybrid Parallel Programming for Blue Gene/P. *Scalable Computing: Practice and Experience*, 12(2):265–274, 2011.

[69] Mads R. B. Kristensen and Brian Vinter. Numerical Python for Scalable Architectures. In *Fourth Conference on Partitioned Global Address Space Programming Model, PGAS'10*. ACM, 2010.

[70] Jonas Latt. Channel flow past a cylinderical obstacle, using a LB method, 2006.

[71] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.

[72] C. E. Leiserson. Fat-Trees: Universl Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, c-34(10), 1985.

[73] D.B. Loveman. High Performance Fortran. *IEEE Parallel & Distributed Technology: Systems & Applications*, 1(1):25, 1993.

[74] MATLAB. *version 7.10.0 (R2010a)*. The MathWorks Inc., Natick, Massachusetts, 2010.

[75] H. Meuer, E. Strohmaier, J. J. Dongarra, and H. D. Simon. Top500 supercomputer sites, Nov 2009.

[76] J. J. Mortensen, L. B. Hansen, and K. W. Jacobsen. Real-space grid implementation of the projector augmented wave method. *Physical Review B*, 71(3):035109, 2005.

[77] Chris J. Newburn, Byoungro So, Zhenying Liu, Michael Mccool, Anwar Ghuloum, Stefanus Du Toit, Zhi Gang Wang, Zhao Hui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu, and Dan Zhang. Intel ' s Array Building Blocks : A Retargetable , Dynamic Compiler and Embedded Language. *Symposium A Quarterly Journal In Modern Foreign Literatures*, pages 1–12, 2011.

[78] J. Nieplocha and M. Krishnan. High Performance Remote Memory Access Comunications: The ARMCI Approach. *International Journal of High Performance Computing and Applications*, 20:2006, 2005.

[79] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2), 1996.

[80] Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[81] Null Nvidia. NVIDIA Corporation, 2010.

[82] Travis E. Oliphant. Python for Scientific Computing. *Computing in Science and Engineering*, 9:10–20, 2007.

[83] Khronos Opencl, Working Group, and Aaftab Munshi. OpenCL Specification. *ReVision*, pages 1–377, 2010.

[84] Ruud Van Der Pas. An Introduction Into OpenMP. *ACM SIGARCH Computer Architecture News*, 34(5):1–82, 2005.

[85] Fernando Pérez and Brian E. Granger. IPython: a System for Interactive Scientific Computing. *Comput. Sci. Eng.*, 9(3):21–29, may 2007.

[86] R. Pozo and B. Miller. SciMark 2.0, 12 2002.

[87] Loïc Prylli and Bernard Tourancheau. Fast Runtime Block Cyclic Data Redistribution on Multiprocessors. *J. Parallel Distrib. Comput*, 45(1):63–72, 1997.

[88] Gautam Shah, Jarek Nieplocha, Jamshed H. Mirza, Chulho Kim, Robert J. Harrison, Rama Govindaraju, Kevin J. Gildea, Paul DiNicola, and Carl A. Bender. Performance and Experience with LAPI - a New High-Performance Communication Library for the IBM RS/6000 SP. In *IPPS/SPDP*, pages 260–266, 1998.

[89] Tom Shanley. *Infiniband*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[90] Fengguang Song, Asim YarKhan, and Jack Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, New York, NY, USA, 2009. ACM.

[91] Carlos Sosa. *IBM System Blue Gene Solution: Blue Gene/P Application Development*. INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION, 2008.

[92] B. A. Stern. Interactive Data Language. 2000.

[93] Volker Strumpen and Thomas L. Casavant. Exploiting Communication Latency Hiding for Parallel Network Computing: Model and Analysis. In *Proc. PDS'94*, pages 622–627. IEEE, 1994.

[94] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency: Pract. Exper.*, 2(4):315–339, 1990.

[95] IBM BLUE GENE TEAM. Overview of the IBM Blue Gene/P project. *IBM Journal of Research and Development*, 52, 2008.

[96] R. Development Core Team. *R: A Language and Environment for Statistical Computing*. Vienna, Austria, 2011.

[97] N. Travinin Bliss and J. Kepner. pMATLAB Parallel MATLAB Library. *International Journal of High Performance Computing Applications*, 21(3):336–359, 2007.

[98] Stéfan van der Walt. NumPy: lock 'n load, 2008.

[99] Guido van Rossum. Python Programming Language. Python Software Foundation, 2009.

[100] T. Warburton. Lecture 24: Brief introduction to block LU factorization and parallel implementation., 2011.

[101] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. Productivity and performance using partitioned global address space languages. In *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32, New York, NY, USA, 2007. ACM.

[102] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: a high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11–13):825–836, 1998.

Appendix A

Publications

A.1 GPAW Optimized for Blue Gene/P using Hybrid Programming

Mads Ruben Burgdorff Kristensen, Hans Henrik Happe, and Brian Vinter. GPAW Optimized for Blue Gene/P using Hybrid Programming
In Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing (IPDPS '09). IEEE Computer Society, Washington, DC, USA, 1-6.

GPAW optimized for Blue Gene/P using hybrid programming

Mads Ruben Burgdorff Kristensen
eScience Centre
University of Copenhagen
Denmark

Hans Henrik Happe
eScience Centre
University of Copenhagen
Denmark

Brian Vinter
eScience Centre
University of Copenhagen
Denmark

***Abstract*—In this work we present optimizations of a Grid-based projector-augmented wave method software, GPAW [1] for the Blue Gene/P architecture. The improvements are achieved by exploring the advantage of shared and distributed memory programming also known as hybrid programming. The work focuses on optimizing a very time consuming operation in GPAW, the finite-different stencil operation, and different hybrid programming approaches are evaluated. The work succeeds in demonstrating a hybrid programming model which is clearly beneficial compared to the original flat programming model. In total an improvement of 1.94 compared to the original implementation is obtained. The results we demonstrate here are reasonably general and may be applied to other finite difference codes.**

I. Introduction

GPAW[1] is a simulation software which simulates many-body systems at the sub-atomic level. GPAW is primarily used by physicists and chemists to investigate the electronic structure, principally the ground state, of many-body systems. A significant part of a GPAW computation consists of a distributed finite-difference operation. The main object of this paper is to optimize this finite-difference operation on the Blue Gene/P[2] (BGP).

BGP, like most popular HPC hardware, consists of multiple shared-memory computation nodes. A hybrid programming paradigm may therefore be explored when targeting the BGP architecture. Unfortunately, it is not trivial to obtain good performance when combining threads and MPI[3]. It is often the case that the sole use of MPI outperforms a combination of OpenMP/Pthread and MPI when computing on clusters of SMP computation nodes[4], [5], [6].

II. GPAW

GPAW is a real-space grid implementation of the projector augmented wave method[7]. It uses uniform real-space grids and the finite-difference approximation for the density functional theory calculations.

A central part of density functional theory and a very time consuming task in GPAW, is to solve Poisson and Kohn-Sham equations. Both equations rely on finite-difference operations when solved by GPAW. When solving the Poisson equation, a finite-difference stencil is applied to the electrostatic potential of the system. When solving the Kohn-Sham equation, a finite-difference stencil is applied to all wave-functions in the system. Both the electron density and the wave-functions are represented by real-space grids. A system typically consists of one electron density and thousands of wave-functions. The number of wave-functions in a system depends on the number of valence electrons in the system. For every valence electron there may be up to two wave-functions.

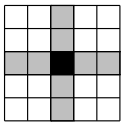


Fig. 1. A stencil operation on a 2D grid.

The computational magnitude of a GPAW simulation depends mainly on three factors: The world size, simulation system resolution and the number of valence electrons. The world size and resolution determine the dimensions of the real-space grids and the number of valence electrons determines the number of real-space grids.

A user is typically more interested in adding valence electrons to the simulation than to increase the size or resolution of the world. The real-space grid size will ordinary be between 100^3 to 200^3 where as the total number of real-space grids will be greater than thousand.

A. Finite-difference

A stencil operation updates a point in a grid based on the surrounding points. A typical 2D example is illustrated in Figure 1 where points are updated based on the two nearest points in all four directions.

The finite-difference methods used in GPAW are stencil operations on the real-space grids (3D arrays). The stencil operation used is a linear combination of a point's two nearest neighbors in all six directions and itself. The stencil operations do normally use periodic boundary condition but that is not always the case.

If we look at the real-space grid A and a predefined list of constants C, a point $A_{x,y,z}$ is computed like this:

$$\begin{aligned} A'_{x,y,z} \quad &= C_1 A_{x,y,z} + C_2 A_{x-1,y,z} + C_3 A_{x+1,y,z} + \\ &C_4 A_{x-2,y,z} + C_5 A_{x+2,y,z} + C_6 A_{x,y-1,z} + \\ &C_7 A_{x,y+1,z} + C_8 A_{x,y-2,z} + C_9 A_{x,y+2,z} + \\ &C_{10} A_{x,y,z-1} + C_{11} A_{x,y,z+1} + \\ &C_{12} A_{x,y,z-2} + C_{13} A_{x,y,z+2} \end{aligned}$$

TABLE I
HARDWARE DESCRIPTION OF A BLUE GENE/P NODE

Node CPU	Four PowerPC 450 cores
CPU frequency	850 MHz
L1 cache (private)	64KB per core
L2 cache (private)	Seven stream prefetching
L3 cache (shared)	8MB
Main memory	2GB
Main memory bandwidth	13.6GB/s
Peak performance	13.6 Gflops/node
Torus bandwidth	$6 \times 2 \times 425$MB/s = 5.1GB/s

III. BLUE GENE/P

Blue Gene/P consists of a number of nodes interconnected with three independent networks: a 3D torus network, a collective tree structured network, and a global barrier network. All point-to-point communication goes through the torus network and every node is equipped with a direct memory access (DMA) engine to offload torus communication from the CPUs. The collective tree structured network is used for collective operation like the MPI *reduce* operation and the global barrier network is used for barriers.

Table I is a brief description of a BGP node. One thing to highlight is the ratio between the speed of the CPU-cores and the main memory. Since the CPU-cores are relatively slow and the main memory is relatively fast compared to today's standard, the performance of the main memory is not as far behind the CPU as usually. Furthermore, the torus bandwidth is only three times lower than the main memory if all six connections are used. The von Neumann bottleneck associated with main memory and network is therefore reduced.

The CPU-cores can be utilized by normal SMP approaches like pthread or OpenMP, with the limitation that BGP only supports one thread per CPU-core. The BGP addresses the problem of utilizing multiple CPU-cores by supporting a virtual partition of the nodes. From the programmers point of view the four CPU-cores would then look like four individual nodes with each 512MB of main memory. This virtual partitioning is called virtual mode.

A. MPI

BGP implements the MPICH2 library which comply with the MPI-2 specification[8]. MPI-2 specifies different levels of threaded communication. BGP supports the fully thread-safe mode called `MULTIPLE` which allows any thread to call the MPI library at any time. Since there is an overhead associated with `MULTIPLE` (e.g. locks), it is also possible to use the more restricted `SINGLE` mode, which do not allow concurrent calls to MPI.

The MPICH2 implementation is tailored to utilize the BGP's DMA engine which means that non-blocking MPI communication is handled asynchronously with minimum CPU involvement.

BGP supports the `MPI_Cart_create` function which tells BGP to reorder the MPI ranks in order to match the torus network. We make use of this function in all the following.

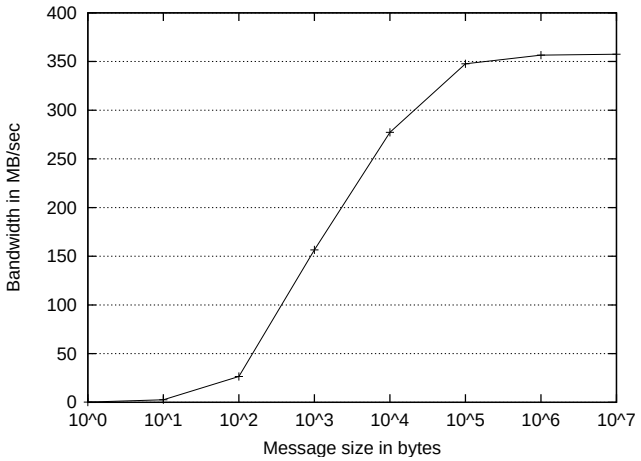


Fig. 2. A bandwidth graph showing how the message size influence the bandwidth. In this experiment, one MPI message is send between two neighboring BGP nodes.

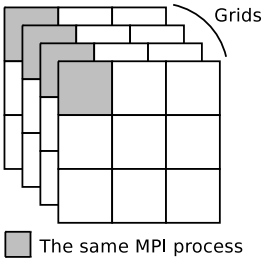


Fig. 3. Four 2D grids distributed over nine processes.

To investigate how much the message size influence point-to-point bandwidth, we have performed an experiment in which one MPI message is send between two neighboring BGP nodes (Figure 2). The result of the experiment clearly shows that in order to maximize the bandwidth, a message size greater than 10^5 bytes is needed, while half the asymptotic bandwidth is achieved at approximate 10^3 bytes.

IV. THE GPAW IMPLEMENTATION

GPAW is implemented using C and Python. The intention is that the users of GPAW should write the model description in Python and then call C and Fortran functions from within Python. It is in this context a user would apply the C implemented finite-difference operation on one or more real-space grids.

The parallel version of GPAW uses MPI in a flat programming model and the parallelization is done by simple domain decomposition of every real-space grid in the simulation. That is, every MPI process gets the same subset of *every* real-space grid in the simulation. This is important because some part of the GPAW computation, like the orthogonalization of wave-functions, requires the same subset of every real-space grid in the simulation. This is illustrated in Figure 3 with 2D real-space grids instead of 3D grids.

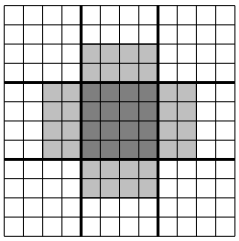


Fig. 4. 2D grid distributed over nine processes. A process needs some of its neighbor's surface points, to compute its own surface points.

The grids are simply divided into a number of quadrilaterals matching the number of available MPI processes. If no user-defined domain decomposition is present, GPAW will try to minimize the aggregated surface of the quadrilaterals. A real-space grid is represented as a three dimensional array where every point in the grid can be a real or complex number (8 or 16 bytes)

A. Distributed Finite-difference

Generally, it should be easy to obtain good scalability for a distributed finite-difference operation since computation grows faster than communication. If we look at a 3D grid of size $n \times n \times n$ the aggregated computation is $O\left(n^3\right)$ where as the aggregated communication is only $O\left(n^2\right)$. The operation should scale very well when n grows at the same rate as the number of CPUs.

In GPAW, however, scalability is very hard to obtain since the grid size will ordinarily not exceed 200^3. Furthermore, since GPAW requires that every MPI process gets the same subset of every grid, it is hard to take advantage of the fact that the number of grids grows at the same pace as the CPUs.

One feature in GPAW which makes it easier to parallelize, is the fact that the input grid and the output grid used in the finite-difference operation is always two separate grids. We need, therefore, not consider the order in which the grid-points are computed.

Applying a finite-difference operation on a grid involves all MPI processes. It is possible for an MPI process to compute most of the points in the sub-grid assigned to it. However, points near the surface of the sub-grid, *surface points*, are dependent on remote points located in neighboring MPI processes. This dependency is illustrated in Figure 4.

The straightforward approach, and the one used in GPAW, for making remote points available, is to exchange the surface points between neighboring MPI processes before applying the finite-difference operation. The serialized communication pattern looks like this:

1) Exchange surface points in the first dimension.
2) Exchange surface points in the second dimension.
3) Exchange surface points in the third dimension.
4) Apply the finite-difference operation.

V. Optimizations

In order to make GPAW run faster on the BGP, we have explored different optimizations. Optimizations which have been beneficial, will be discussed in this section.

The most obvious optimization is to exchange surface elements simultaneously in all three dimensions, by using the following non-blocking communication pattern:

1) Initiate the exchange of surface points in all three dimensions.
2) Wait for all exchanges to finish.
3) Apply the finite-difference operation.

The idea is to fully utilize the torus network in all six directions simultaneously, see Table I.

Another important performance aspect is how to map the distributed real-space grids onto the physical network topology. The 3D torus network is used for point-to-point communication in MPI, thus it is the network, we should attempt to map the distributed real-space grids onto. Since the grids have the same number of dimensions as the torus network, and since the finite-difference operation may use periodic boundary condition, a torus topology is a perfect match to our problem. However, the BGP requires a partition with 512 or more nodes to form a torus topology. A partition under 512 nodes can only form a mesh topology.

A. Multiple real-space grids

Double buffering and communication batching are two techniques which can improve the performance of the finite-difference operation. Both techniques requires multiple real-space grids but the finite-difference operation is typically applied on thousands of real-space grids.

Double buffering

Double buffering is a technique which makes it possible to overlap communication and computation. The following communication pattern illustrates how:

1) Initiate the exchange of surface points in all three dimensions for the first grid.
2) Initiate the exchange of surface points in all three dimensions for the second grid.
3) Wait for all exchanges of the first grid to finish.
4) Apply the stencil operation on the first grid.
5) Initiate the exchange of surface points in all three dimensions for the third grid.
6) Wait for all exchanges of the second grid to finish.

The performance gain is dependent on the ability of the MPI library and the underlying hardware to process non-blocking send and receive calls. On the BGP, progress in non-blocking send and receive calls will be maintained by the DMA engine and increased performance is therefore expected.

Batching

An way to ensure critical packet size is to pack real-space grids into batches; inspired by the message size experiment (Figure 2).

Continuously dividing the grids between more and more MPI processes reduces the number of surface points in a single sub-grid. That is, at some point the amount of data send by a single MPI call will be reduced to a size in which the MPI

overhead and network latency will dominate the communication overhead. The idea is to send a batch of surface points in each MPI call, instead of sending surface points, individually. This will reduce the communication overhead considerably, as the size of the sub-grids decreases. The number of grids packed together in this way, we call the *batch-size*.

When using double buffering, it is important to allow the CPUs to start computing as soon as possible. Combining a large batch-size with double buffering will therefore introduce a penalty as the initial surface points exchange cannot be hidden. One approach to minimize this penalty, is to increase the batch-size continuously in the initial stage. For instance a batch-size of 128 could be reduced to 64 in the initial exchange.

VI. Programming approaches

Different approaches exist when combining threads and MPI. To preserve control we have chosen to handle the threading manually in pthread.

The following is a description of different programming approaches that we have investigated. Every programming approach except the **Flat original** uses the optimizations described in section V.

- **Flat original** is the approach originally used in GPAW. It uses the BGP's virtual mode, where the four CPU-cores are treated as individual nodes, to utilize all four CPU-cores and it is therefore not necessary to modify anything to support the BGP architecture.
- **Flat optimized** is an optimized version of the original approach and just like the **Flat original** it uses the virtual mode.
- **Hybrid multiple** does not use the virtual mode. Instead, one hardware thread per CPU-core is spawned. Every thread handles its own inter-node communication. The node will distribute the real-space grids between its four CPU-cores, not by dividing the grids into smaller pieces but by assigning different grids to every CPU-core. Because of this no synchronization is needed until all grids are computed, the synchronization penalty is therefore constant. This way of exploiting multiple grids is the main advantage of this approach.
- **Hybrid master-only** also spawns one thread per CPU-core, but only one thread, the *master thread*, handles inter-node communication. Since we have to synchronize between every grid-computation, each grid-computation will be divided between the four CPU-cores. The synchronization penalty thus become proportional to the number of grids. On the other hand, this approach does work in `SINGLE` MPI-mode and the overhead associated with `MULTIPLE` is therefore avoided.

VII. Results

A benchmark of each implementation has been executed on the Blue Gene/P. 16384 CPU-cores or 4096 nodes or 4 racks were made available to us. Every benchmark graph compares the different programming approaches of the finite-difference operation in GPAW and a periodic boundary condition is used in all cases.

Figure 5 is a classic speedup graph comparing every implemented approach with a sequential execution. It is a relatively small job containing only 32 real-space grids. But because of the memory demand, it is not possible to have more than 32 grids running on a single CPU-core.

The result clearly show that the best scaling and running time is obtained with **Flat optimized** and **Hybrid multiple** both using a batch-size of 8 grids. Since the job only consists of 32 grids a batch-size of 8 is the maximum if all four CPU-cores should be used. Another interresting observation is that the advantage of batching is greater in **Hybrid multiple** than in **Flat optimized**. This indicates that if a job consist of more grids, the **Hybrid multiple** approach may become faster than **Flat optimized**.

A. Multiple real-space grids

As the number of grids grow there is a corresponding linear growth in the computation required in the finite-difference operation. It is therefore possible to create a Gustafson graph by increasing the number of grids in the same rate as the number of CPU-cores (Figure 6). It is important to note that the required communication per node increases faster than the needed computation; this is due to the increased surface size associated with the additional partitioning of the grids. To illustrate this communication increase, the right scale in Figure 6 shows the needed communication per node for **Flat optimized** and **Hybrid multiple** respectively.

At 512 CPU-cores **Hybrid multiple** is faster than **Flat optimized**. The main reason is the difference in the needed communication. **Flat optimized** divides the grids four times more than the **Hybrid multiple**. We did not see this effect in the speedup graph, Figure 5, because of the small number of grids. Furthermore, **Hybrid multiple** is better to exploit an increase in grids because of the thread synchronization overhead. The overhead is small and constant, but since the total running time is very small for 32 grids (9 milliseconds with 2048 CPU-cores), the impact of the synchronization overhead is drastically reduced when the number of grids, and thereby the total running time, is increased.

To investigate the scalability of a large job with many real-space grids, we have made a speedup graph beginning at 1k CPU-cores, which allows for a 2816 grid job (Figure 7). Again **Hybrid multiple** has the best performance - going from 1k to 16k CPU-cores gives a speedup of approximately 16.5 compared to **Flat original**. Comparing **Hybrid multiple** with itself, we have a speedup of 12 where 16 would be linear but unobtainable due to the increase in the needed communication.

To further investigate the performance difference between **Hybrid multiple** and **Flat optimized**, we have made a small experiment. We modifies **Flat optimized** to statically divide the real-space grids into four sub-groups. It is now possible for all four CPU-cores to work on its own sub-group and the real-space grids will be divided into the same level as in **Hybrid multiple**. The only difference between the two approaches is

that **Flat optimized** uses BGP's virtual mode and **Hybrid multiple** uses threads. It should be noted, however, that in a real GPAW computation this modification does not work, since GPAW requires that every MPI process gets the same subset of every real-space grid, see section IV. The experiment is not included in any of the graphs since its performance is identical with the **Hybrid multiple**. Because of the identical performance, we find it reasonable to conclude that the level of real-space partitioning is the sole reason for the performance difference between **Hybrid multiple** and the non-modified **Flat optimized**.

VIII. Conclusions

Overall this work has managed to improve the performance of a domain specific finite-difference code when scaling to very large systems. The primary improvements are obtained through the introduction of asynchronous communication which, even in a well balanced system such as the Blue Gene, efficiently improves processor utilization. Furthermore, two hybrid programming approaches have been explored: the hybrid multiple and the master-only approach.

The hybrid programming approach, in which inter-node communication is handled individually by every thread, has shown a positive impact on the performance. By allowing every thread to handle its own inter-node communication, the overhead for thread synchronization remains constant and the application becomes faster than the non-hybrid version.

On the other hand, the alternative hybrid programming approach, in which one thread handles the inter-node communication on behalf of all threads in the process, cannot compete with the non-hybrid version. That is explained by the overhead that is introduced by thread synchronization which grows proportional to the number of grids in the computation.

When comparing our fastest implementation compared to the original implementation, the hybrid programming approach combined with the latency-hiding techniques is 94% faster at 16384 CPU-cores. Translated into utilization this means that CPU utilization grows from 36% to 70%.

While latency-hiding is the primary factor for the improvement we observe, the hybrid implementation is still 10% faster than the non-hybrid approach.

A. Further work

Overall we are satisfied with the performance of the new implementation of the finite-difference operation, still a lot of work remains if the entire GPAW computation should utilize latency-hiding and hybrid programming. It may not be worth the hard work that is needed to rewrite most of GPAW, but it is our expectation that an overall performance gain as the one demonstrated in this work may be obtained for the application overall.

Acknowledgments

The authors would like to thank the GPAW team at the Technical University of Denmark in particular Jens J. Mortensen and Marcin Dulak. Furthermore we would like to thank Argonne National Laboratory for giving us access to the Blue Gene/P.

References

[1] J. J. Mortensen, L. B. Hansen, and K. W. Jacobsen, "Real-space grid implementation of the projector augmented wave method," *Physical Review B*, vol. 71, no. 3, p. 035109, 2005.

[2] I. B. G. TEAM, "Overview of the IBM Blue Gene/P project," *IBM Journal of Research and Development*, vol. 52, 2008.

[3] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI Portable Parallel Programming with the Message Passing Interface*. The MIT Press, 1994.

[4] D. S. Henty, "Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling," *Supercomputing, ACM/IEEE 2000 Conference*, pp. 10–10, Nov. 2000.

[5] M. Hipp and W. Rosenstiel, *Parallel Hybrid Particle Simulations Using MPI and OpenMP*. Springer-Verlag Berlin Heidelberg, 2004, pp. 189–197.

[6] B. Vinter and J. M. Bjørndalen, "A Comparison of Three MPI Implementations," in *Communicating Process Architectures 2004*, I. R. East, D. Duce, M. Green, J. M. R. Martin, and P. H. Welch, Eds., sep 2004, pp. 127–136.

[7] P. E. Blochl, "Projector augmented-wave method," *Phys. Rev. B*, vol. 50, no. 24, pp. 17 953–17 979, Dec 1994.

[8] W. Gropp, S. Huss-Lederman, A. Limsdaine, E. Lusk, W. Saphir, and M. Snir, *The Complete Reference: Volume 2, the MPI-2 Extensions*. MIT Press, 1998.

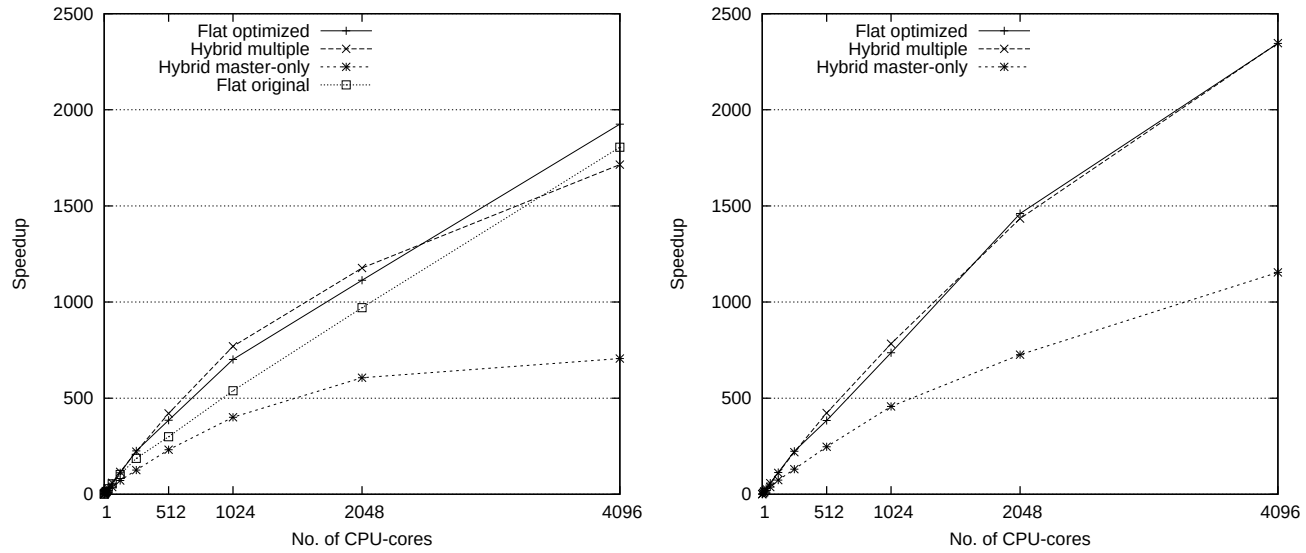


Fig. 5. Speedup of the finite-difference operation. The job consist of only 32 real-space grids all with a size of 144^3. In the left graph batching is disabled and in the right graph batching is enabled using a batch-size of 8.

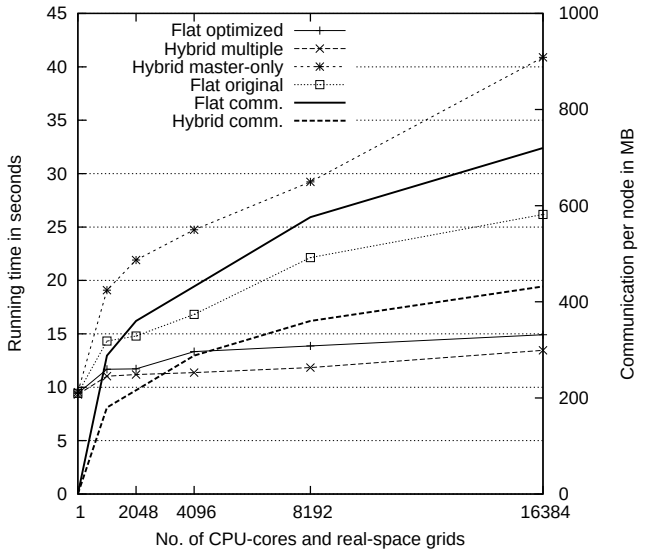


Fig. 6. A Gustafson graph showing the running time of the finite-difference operation when the number of real-space grids is increasing in the same rate as the number of CPU-cores - one grid per CPU-core. The grid size are 192^3 and the best batch-size has been found for every number of CPU-cores. The right scale shows the needed communication per node.

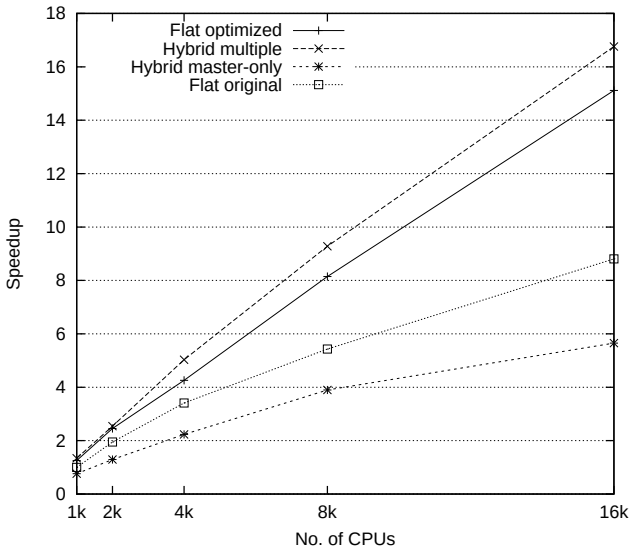


Fig. 7. A Speedup graph starting at 1024 CPU-cores running the finite-difference operation; every approach is compared to the original approach at 1024 CPU-cores. The job consists of 2816 real-space grids all size of 192^3, and the best batch-size has been found for every number of CPU-cores.

A.2 Hybrid Parallel Programming for Blue Gene/P

M. Kristensen, H. Happe, and B. Vinter, Hybrid Parallel Programming for Blue Gene/P
Scalable Computing: Practice and Experience, vol. 12, no. 2, 2011. ISSN 1895-1767.

HYBRID PARALLEL PROGRAMMING FOR BLUE GENE/P

MADS R. B. KRISTENSEN , HANS H. HAPPE , AND BRIAN VINTER*

Abstract. The concept of massively parallel processors has been taken to the extreme with the introduction of the BlueGene architectures from IBM. With hundreds of thousands of processors in one machine the parallelism is extreme, but so are the techniques that must be applied to obtain performance with that many processors. In this work we present optimizations of a Grid-based projector-augmented wave method software, GPAW, for the Blue Gene/P architecture. The improvements are achieved by exploring the advantage of shared and distributed memory programming also known as hybrid programming and blocked communication to improve latency hiding. The work focuses on optimizing a very time consuming operation in GPAW, the stencil operation, and different hybrid programming approaches are evaluated. The work succeeds in demonstrating a hybrid programming model, which is clearly beneficial compared to the original flat programming model. In total an improvement of 1.94 compared to the original implementation is obtained. The results we demonstrate here are reasonably general and may be applied to other stencil codes.

Key words. GPAW, HPC, Hybrid-programming, Multicore platforms

1. Introduction. Grid Based Projector Augmented Wave (GPAW)[8] is a simulation software, which simulates many-body systems at the sub-atomic level. GPAW is primarily used by physicists and chemists to investigate the electronic structure, principally the ground state, of many-body systems. The GPAW users often have a desire to increase the system size and resolution to the point where the simulation time escalates to weeks and sometimes even months. A massively parallel implementation of GPAW, which is able to fully utilize a supercomputer, is therefore highly desirable.

The performance profile of GPAW dependence almost entirely on the electronic structures that are being simulated. Therefore, it is difficult to measure the general performance of GPAW. However, a significant part of any GPAW computation consists of a distributed stencil operation. Thus an optimization of this stencil operation will result in an improvement of the general performance of GPAW. The main object of this paper is to optimize the stencil operation for the Blue Gene/P[9] (BGP) architecture.

The current trend in HPC hardware is towards systems of shared-memory computation nodes. The BGP also follows this trend and consists of four CPU-cores per node. Furthermore, it is quite possible that future versions of the Blue Gene architecture will consists of even more CPU-cores per node.

To exploit the memory locality in shared-memory computation nodes a paradigm that combines shared and distributed memory programming may be of interest. The idea is to avoid communication between CPU-cores on the same node. Unfortunately, it is not trivial to obtain good performance when combining shared-memory programming with distributed memory programming. Even though inter-CPU communication is avoided, it is often the case that the sole use of MPI[5] outperforms a combination of threads and MPI when computing on clusters of shared-memory computation nodes[6, 7, 10].

We evaluate two different hybrid programming approaches. One approach in which inter-node communication is handled individually by every thread and another approach in which one thread handles the inter-node communication on behalf of all the other threads in a node. The work shows that, on the Blue Gene/P, the first approach is clearly superior the latter. In [3] the authors concludes that, on a well

*Niels Bohr Institute, Copenhagen, Denmark. {madsbk, happe, vinter}@nbi.dk

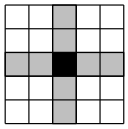


FIG. 2.1. *A stencil operation on a 2D grid.*

balanced system, a loop level parallelization approach, corresponding to our second hybrid approach, is unfavorably compared to a strict MPI implementation. Our first hybrid approach was developed on the basis of that conclusion.

2. GPAW. GPAW is a real-space grid implementation of the projector augmented wave method[2]. It uses uniform real-space grids and the finite-difference approximation for the density functional theory calculations.

A central part of density functional theory and a very time consuming task in GPAW, is to solve Poisson and Kohn-Sham equations. Both equations rely on stencil operations when solved by GPAW. When solving the Poisson equation, a stencil is applied to the electrostatic potential of the system. When solving the Kohn-Sham equation, a stencil is applied to all wave-functions in the system. Both the electron density and the wave-functions are represented by real-space grids. A system typically consists of one electron density and thousands of wave-functions. The number of wave-functions in a system depends on the number of valence electrons in the system. For every valence electron there may be up to two wave-functions.

The computational magnitude of a GPAW simulation depends mainly on three factors: The world size, simulation system resolution and the number of valence electrons. The world size and resolution determine the dimensions of the real-space grids and the number of valence electrons determines the number of real-space grids.

A user is typically more interested in adding valence electrons to the simulation than to increase the size or resolution of the world. The real-space grid size will ordinary be in the interval 100^3 to 200^3 where as the total number of real-space grids will be greater than thousand.

2.1. Stencil Operation. A stencil operation updates a point in a grid based on the surrounding points. A typical 2D example is illustrated in Fig. 2.1 where points are updated based on the two nearest points in all four directions.

Stencil operations on the real-space grids (3D arrays) are used for the finite-difference approximation in GPAW. The stencil operation used is a linear combination of a point's two nearest neighbors in all six directions and itself. The stencil operations do normally use periodic boundary condition but that is not always the case.

If we look at the real-space grid A and a predefined list of constants C, a point $A_{x,y,z}$ is computed like this:

$$\begin{aligned} A'_{x,y,z} \quad = \quad & C_1 A_{x,y,z} + C_2 A_{x-1,y,z} + C_3 A_{x+1,y,z} + \\ & C_4 A_{x-2,y,z} + C_5 A_{x+2,y,z} + C_6 A_{x,y-1,z} + \\ & C_7 A_{x,y+1,z} + C_8 A_{x,y-2,z} + C_9 A_{x,y+2,z} + \\ & C_{10} A_{x,y,z-1} + C_{11} A_{x,y,z+1} + \\ & C_{12} A_{x,y,z-2} + C_{13} A_{x,y,z+2} \end{aligned}$$

3. Blue Gene/P. Blue Gene/P consists of a number of nodes interconnected with three independent networks: a 3D torus network, a collective tree structured network, and a global barrier network. All point-to-point communication goes through

TABLE 3.1
Hardware description of a Blue Gene/P node

Node CPU	Four PowerPC 450 cores
CPU frequency	850 MHz
L1 cache (private)	64KB per core
L2 cache (private)	Seven stream prefetching
L3 cache (shared)	8MB
Main memory	2GB
Main memory bandwidth	13.6GB/s
Peak performance	13.6 Gflops/node
Torus bandwidth	$6 \times 2 \times 425$MB/s $= 5.1$GB/s

the torus network and every node is equipped with a direct memory access (DMA) engine to offload torus communication from the CPUs. The collective tree structured network is used for collective operation like the MPI *reduce* operation and the global barrier network is used for barriers.

Table 3.1 is a brief description of a BGP node. One thing to highlight is the ratio between the speed of the CPU-cores and the main memory. Since the CPU-cores are relatively slow and the main memory is relatively fast compared to today's standard, the performance of the main memory is not as far behind the CPU as usually. Furthermore, the torus bandwidth is only three times lower than the main memory bus when all six connections are used. The von Neumann bottleneck[1] associated with main memory and network is therefore reduced.

The CPU-cores can be utilized by normal SMP approaches like pthread or OpenMP, with the limitation that BGP only supports one thread per CPU-core. The BGP addresses the problem of utilizing multiple CPU-cores by supporting a virtual partition of the nodes. From the programmers point of view the four CPU-cores would then look like four individual nodes with each 512MB of main memory. This virtual partitioning is called virtual mode.

3.1. MPI. BGP implements the MPICH2 library, which comply with the MPI-2 specification[4]. MPI-2 specifies different levels of threaded communication. BGP supports the fully thread-safe mode called `MULTIPLE` that allows any thread to call the MPI library at any time. Since there is an overhead associated with `MULTIPLE` (e.g. locks), it is also possible to use the more restricted `SINGLE` mode that do not allow concurrent calls to MPI.

The MPICH2 implementation is tailored to utilize the BGP's DMA engine which means that non-blocking MPI communication is handled asynchronously with minimum CPU involvement.

BGP supports the `MPI_Cart_create` function, which tells BGP to reorder the MPI ranks in order to match the torus network. We make use of this function in all the following.

To investigate how much the message size influence point-to-point bandwidth, we have performed an experiment in which one MPI message is send between two neighboring BGP nodes (cf. Fig. 3.1). The result of the experiment clearly shows that in order to maximize the bandwidth, a message size greater than 10^5 bytes is needed, while half the asymptotic bandwidth is achieved at approximate 10^3 bytes.

4. The GPAW Implementation. GPAW is implemented using C and Python. The intention is that the users of GPAW should write the model description in Python and then call C and Fortran functions from within Python. It is in this context a user

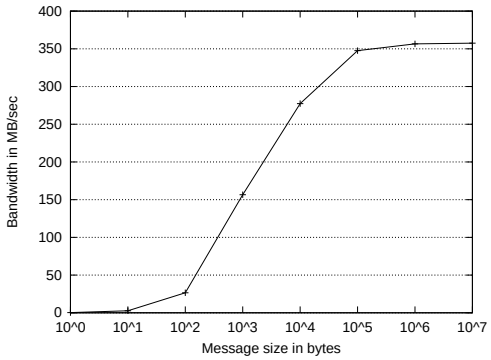


FIG. 3.1. *A bandwidth graph showing how the message size influence the bandwidth. In this experiment, one MPI message is send between two neighboring BGP nodes.*

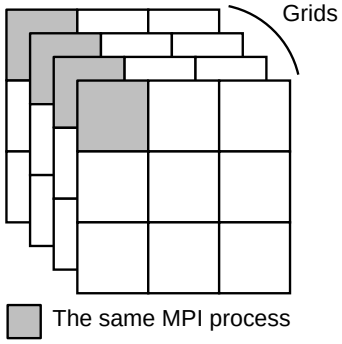


FIG. 4.1. *Four 2D grids distributed over nine processes.*

would apply the C implemented stencil operation on one or more real-space grids.

The parallel version of GPAW uses MPI in a flat programming model and the parallelization is done by simple domain decomposition of every real-space grid in the simulation. That is, every MPI process gets the same subset of *every* real-space grid in the simulation. This is important because some part of the GPAW computation, like the orthogonalization of wave-functions, requires the same subset of every real-space grid in the simulation. This domain decomposition is illustrated in Fig. 4.1 with 2D real-space grids instead of 3D grids.

The grids are simply divided into a number of quadrilaterals matching the number of available MPI processes. If no user-defined domain decomposition is present, GPAW will try to minimize the aggregated surface of the quadrilaterals. A real-space grid is represented as a three dimensional array where every point in the grid can be a real or complex value (8 or 16 bytes)

4.1. Distributed Stencil Operation. Generally, it should be easy to obtain good scalability for a distributed stencil operation since computation grows faster than communication. If we look at a 3D grid of size $n \times n \times n$ the aggregated computation is $O\left(n^3\right)$ where as the aggregated communication is only $O\left(n^2\right)$. The operation should scale very well when n grows at the same rate as the number of CPUs. In GPAW, however, scalability is very hard to obtain since the grid size will ordinarily not exceed 200^3. Thus, the n is smaller than 200 even when parallelizing over thousands of CPUs.

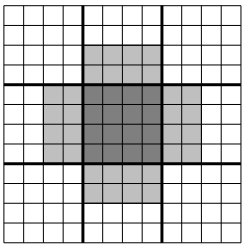


FIG. 4.2. *2D grid distributed over nine processes. A process needs some of its neighbor's surface points, to compute its own surface points.*

The fact that the number of independent grids grows linearly with the number of valence electrons that a simulated would normally make the problem embarrassingly parallel. Each MPI process could compute a whole grid without the need of any communication, since no communication between grids is required in GPAW. However, this is not possible because GPAW requires that every MPI process gets the same subset of every grid (cf. Fig. 4.1).

One feature in GPAW, which makes it easier to parallelize, is the fact that the input grid and the output grid used in the stencil operation is always two separate grids. We need therefore not consider the order in which the grid-points are computed.

Applying a stencil operation on a grid involves all MPI processes. It is possible for an MPI process to compute most of the points in the sub-grid assigned to it. However, points near the surface of the sub-grid, *surface points*, are dependent on remote points located in neighboring MPI processes. This dependency is illustrated in Fig. 4.2.

The straightforward approach, and the one used in GPAW, for making remote points available, is to exchange the surface points between neighboring MPI processes before applying the stencil operation. The serialized communication pattern looks like this:

1. Exchange surface points in the first dimension.
2. Exchange surface points in the second dimension.
3. Exchange surface points in the third dimension.
4. Apply the stencil operation.

5. Optimizations. In order to make GPAW run faster on the BGP, we have explored different optimizations. In this section, we will discuss the optimizations that have been beneficial for the overall performance.

The most obvious optimization is to exchange surface elements simultaneously in all three dimensions by using the following non-blocking communication pattern:

1. Initiate the exchange of surface points in all three dimensions.
2. Wait for all exchanges to finish.
3. Apply the stencil operation.

The idea is to fully utilize the torus network in all six directions simultaneously, see Table 3.1.

Another important performance aspect is how to map the distributed real-space grids onto the physical network topology. The 3D torus network is used for point-to-point communication in MPI, thus it is the network, we should attempt to map the distributed real-space grids onto. Since the grids have the same number of dimensions as the torus network, and since the stencil operation may use periodic boundary

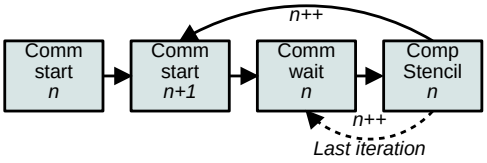


FIG. 5.1. *Flow diagram illustrating double buffering. The n'th iteration is expressed with a n and Comm and Comp stands for communication and computation, respectively. n++ is an iteration to n's successor.*

condition, a torus topology is a perfect match to our problem. However, the BGP requires a partition with 512 or more nodes to form a torus topology. A partition under 512 nodes can only form a mesh topology.

5.1. Multiple Real-space Grids. Double buffering and communication batching are two techniques which can improve the performance of the stencil operation. Both techniques requires multiple real-space grids but the stencil operation is typically applied on thousands of real-space grids.

5.1.1. Double Buffering. Double buffering is a technique that makes it possible to overlap communication and computation. The following communication pattern illustrates how (cf. Fig. 5.1):

1. Initiate the exchange of surface points in all three dimensions for the first grid.
2. Initiate the exchange of surface points in all three dimensions for the second grid.
3. Wait for all exchanges of the first grid to finish.
4. Apply the stencil operation on the first grid.
5. Initiate the exchange of surface points in all three dimensions for the third grid.
6. Wait for all exchanges of the second grid to finish.

The performance gain is dependent on the ability of the MPI library and the underlying hardware to process non-blocking send and receive calls. On the BGP, progress in non-blocking send and receive calls will be maintained by the DMA engine and increased performance is therefore expected.

5.1.2. Batching. An way to obtain critical packet size is to pack real-space grids into batches; inspired by the message size experiment (cf. Fig. 3.1).

Continuously dividing the grids between more and more MPI processes reduces the number of surface points in a single sub-grid. That is, at some point the amount of data send by a single MPI call will be reduced to a size in which the MPI overhead and network latency will dominate the communication overhead. The idea is to send a batch of surface points in each MPI call, instead of sending surface points, individually. This will reduce the communication overhead considerably, as the size of the sub-grids decreases. The number of grids packed together in this way, we call *batch-size*.

When using double buffering, it is important to allow the CPUs to start computing as soon as possible. Combining a large batch-size with double buffering will therefore introduce a penalty as the initial surface points exchange cannot be hidden. One approach to minimize this penalty, is to increase the batch-size continuously in the initial stage. For instance a batch-size of 128 could be reduced to 64 in the initial exchange. This technique we call *sloped batching*.

l	:	latency
B	:	bandwidth
C	:	computation time of one stencil element
t	:	total stencil size
b	:	batch-size
n	:	number of batch-size increasements initially

$$WaitTime = l + \frac{b}{2^n B} + \sum_{i=1}^{n} \max\left(0, l + \frac{b}{2^{i-1}B} - \frac{Cb}{2^i}\right) + \max\left(0, l + \frac{b}{B} - Cb\right)\left(\frac{t}{b} + 1 - 2^{n+1}\right)$$

FIG. 5.2. *Formula of the amount of time used by waiting on non-hidden communication when using double buffering and sloped batching. The first line represents the initial communication, which can not overlap computation. The second line represents the sloped bashing, in which the block-size is doubled in each iteration and the third line represents the rest of the iterations, in which the block-size remains constant.*

The amount of time used by waiting on non-hidden communication depends on many factors – some related to the runtime system and some related to the implementation. A general expression of the relationship is given in figure 5.2, which can be used to find the optimal batch-size and the optimal number of initial batch-size increasements when doing sloped batching. The CPU overhead associated with a implementation of double buffering and sloped batching is not included in the expression likewise the memory access time associated with the stencil computation is also not included.

6. Programming Approaches. Different approaches exist when combining threads and MPI. To preserve control we have chosen to handle the threading manually in pthread.

The following is a description of different programming approaches that we have investigated. Every programming approach except the Flat-original uses the optimizations described in Sect. 5.

Flat-original is the approach originally used in GPAW. It uses the BGP's virtual mode, where the four CPU-cores are treated as individual nodes, to utilize all four CPU-cores. Therefore, it is not necessary to modify anything to support the BGP architecture.

Flat-optimized is an optimized version of the original approach and just like the Flat-original it uses the virtual mode.

Hybrid-multiple does not use the virtual mode. Instead, one hardware thread per CPU-core is spawned. Every thread handles its own inter-node communication. The node will distribute the real-space grids between its four CPU-cores, not by dividing the grids into smaller pieces but by assigning different grids to every CPU-core. Because of this no synchronization is needed until all grids are computed and the synchronization penalty is therefore constant. This way of exploiting multiple grids is the main advantage of this approach.

Hybrid-master-only also spawns one thread per CPU-core, but only one thread, the *master thread*, handles inter-node communication. Since we have to synchronize between every grid-computation, each grid-computation will be divided between the four CPU-cores. The synchronization penalty thus become

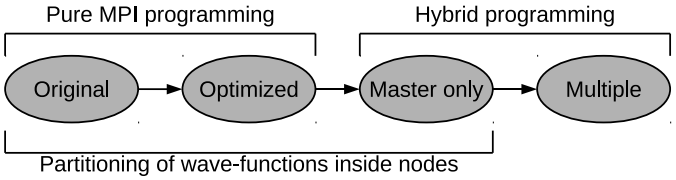


FIG. 6.1. *A illustrates of the relationship between the four programming approaches – going from the Flat-original approach to the Hybrid-multiple approach.*

proportional to the number of grids. On the other hand, this approach does work in `SINGLE` MPI-mode and the overhead associated with `MULTIPLE` is therefore avoided.

Fig. 6.1 illustrates the relationship between the four programming approaches – from the original approach, in which pure MPI programming is used and the wave-functions are partitioned inside the nodes, to the hybrid approach where hybrid programming is used and the wave-functions are shared inside the nodes.

7. Results. A benchmark of each implementation has been executed on the Blue Gene/P (Sec. 3). 16384 CPU-cores or 4096 nodes or 4 racks were made available to us. Every benchmark graph compares the different programming approaches of the stencil operation in GPAW and a periodic boundary condition is used in all cases.

Fig. 7.2 is a classic speedup graph comparing every implemented approach with a sequential execution. It is a relatively small job containing only 32 real-space grids. But because of the memory demand, it is not possible to have more than 32 grids running on a single CPU-core.

The result clearly show that the best scaling and running time is obtained with Flat-optimized and Hybrid-multiple both using a batch-size of 8 grids. Since the job only consists of 32 grids a batch-size of 8 is the maximum if all four CPU-cores should be used. Another interesting observation is that the advantage of batching is greater in Hybrid-multiple than in Flat-optimized. This indicates that if a job consist of more grids, the Hybrid-multiple approach may become faster than Flat-optimized.

7.1. Communication and Computation Profile. The communication and computation profile becomes very important when scaling to a massive number of processes. As the number of MPI processes increases the communication time has a tendency to increase due to network congestion. It is therefore essential that communication is spread evenly between the CPU-core and that the diversity of the communication and computation time is minimized.

Fig. 7.1 is a profile of the Hybrid-multiple approach executing on 1024 CPU-cores. It shows a distinct pattern in which the communication and the computation phase are aligned throughout the execution. From that it is evident that Hybrid-multiple actually do execute in a fairly synchronized manner and no ripple effect of waiting processes is observed.

7.2. Multiple Real-space Grids. As the number of grids grow there is a corresponding linear growth in the computation required in the stencil operation. It is therefore possible to create a Gustafson graph by increasing the number of grids in the same rate as the number of CPU-cores (cf. Fig. 7.3). It is important to note that the required communication per node increases faster than the needed computation. This is due to the increased surface size associated with the additional partitioning

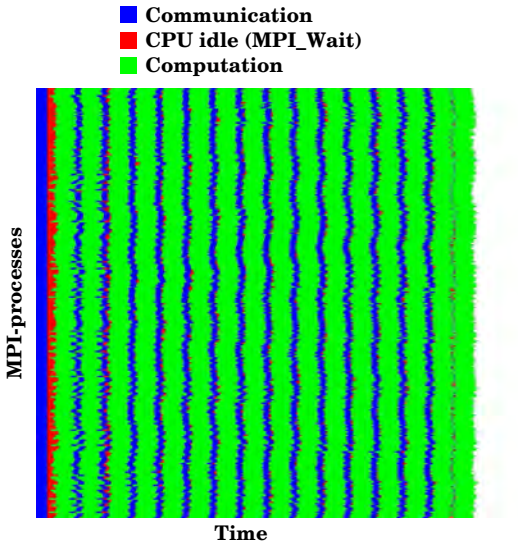


FIG. 7.1. *Profile of the communication and computation pattern when computing 1024 real-space grids on 1024 CPU-cores and the Hybrid-multiple approach is used. A line represents a MPI-process and the length of the line represents the progress of time.*

of the grids. To illustrate this communication increase, the right graph in Fig. 7.3 shows the needed communication per node for Flat-optimized and Hybrid-multiple respectively.

If we, for example, look at a computation of a grid with a size of 192^3 using 1024 nodes, the grid will either be divided between 1024 MPI processes when using Hybrid-multiple or 4096 MPI process when using Flat-optimized. Flat-optimized needs to communicate approximately 140KB more data per node than Hybrid-multiple. Note that this is only for a single real-space grid, the different will grow linearly with the number of grids in the computation.

At 512 CPU-cores Hybrid-multiple is faster than Flat-optimized. The main reason is the difference in the needed communication. Flat-optimized divides the grids four times more than the Hybrid-multiple. We did not see this effect in the speedup graph, Fig. 7.2, because of the small number of grids. Furthermore, Hybrid-multiple is better to exploit an increase in grids because of the thread synchronization overhead. The overhead is small and constant, but since the total running time is very small for 32 grids (9 milliseconds with 2048 CPU-cores), the impact of the synchronization overhead is drastically reduced when the number of grids, and thereby the total running time, is increased.

To investigate the scalability of a large job with many real-space grids, we have made a scalability graph beginning at 1k CPU-cores, which allows for a 2816 grid job (cf. Fig. 7.4). Again Hybrid-multiple has the best performance - going from 1k to 16k CPU-cores gives a speedup of approximately 12.5 where 16 would be linear but unobtainable due to the increase in the needed communication. If we compare the running time of Hybrid-multiple with Flat-original, we see a 94% performance gain at 16384 CPU-cores.

To further investigate the performance difference between Hybrid-multiple and Flat-optimized, we have made a small experiment. We modifies Flat-optimized to statically divide the real-space grids into four sub-groups. It is now possible for

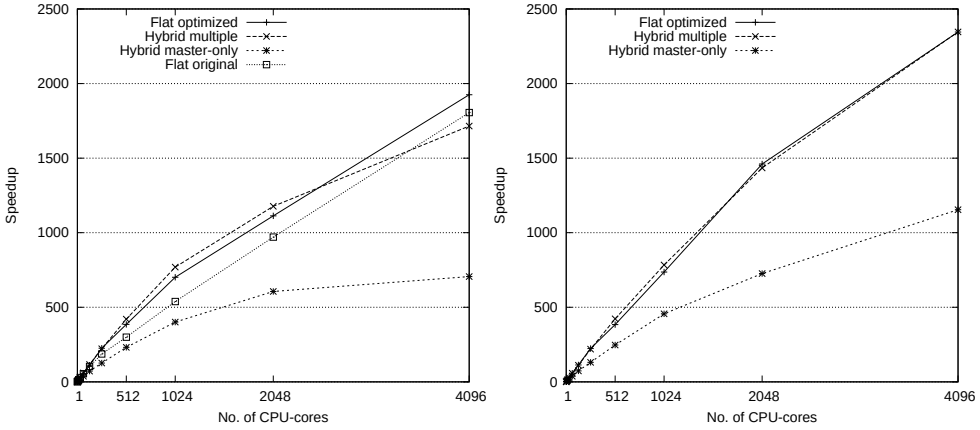


FIG. 7.2. *Speedup of the stencil operation. The job consist of only 32 real-space grids all with a size of* 144^3*. In the left graph batching is disabled and in the right graph batching is enabled using a batch-size of 8.*

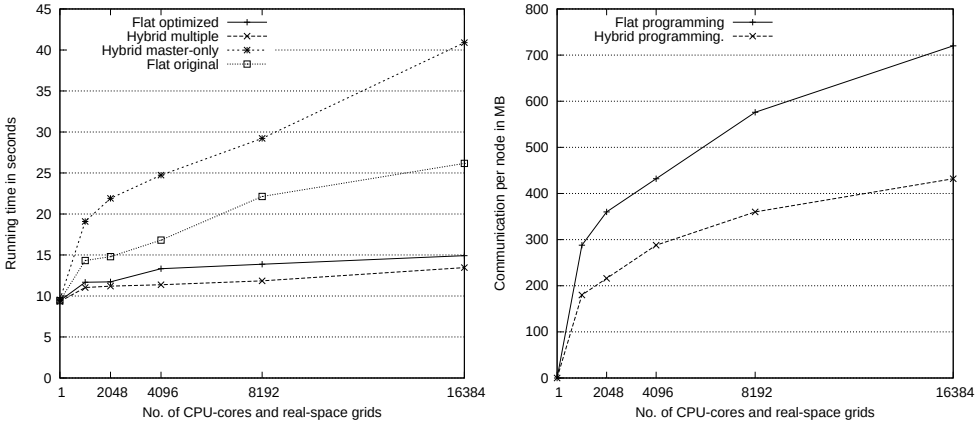


FIG. 7.3. *Gustafson graphs showing the running time of the stencil operation and the needed inter-node communication when the number of real-space grids is increasing in the same rate as the number of CPU-cores - one grid per CPU-core. The left graph shows the running time and the right graph shows the needed inter-node communication. The grid size are* 192^3 *and the best batch-size has been found for every number of CPU-cores.*

all four CPU-cores to work on its own sub-group and the real-space grids will be divided into the same level as in Hybrid-multiple. The only difference between the two approaches is that Flat-optimized uses the virtual mode in Blue Gene/P and Hybrid-multiple uses threads. It should be noted, however, that in a real GPAW computation this modification does not work, since GPAW requires that every MPI process gets the same subset of every real-space grid, see Sect 4. The experiment is not included in any of the graphs since its performance is identical with the Hybrid-multiple. Because of the identical performance, we find it reasonable to conclude that the level of real-space partitioning is the sole reason for the performance difference between Hybrid-multiple and the non-modified Flat-optimized.

8. Conclusions. Overall this work has managed to improve the performance of a domain specific stencil code when scaling to a very high degree of parallelism.

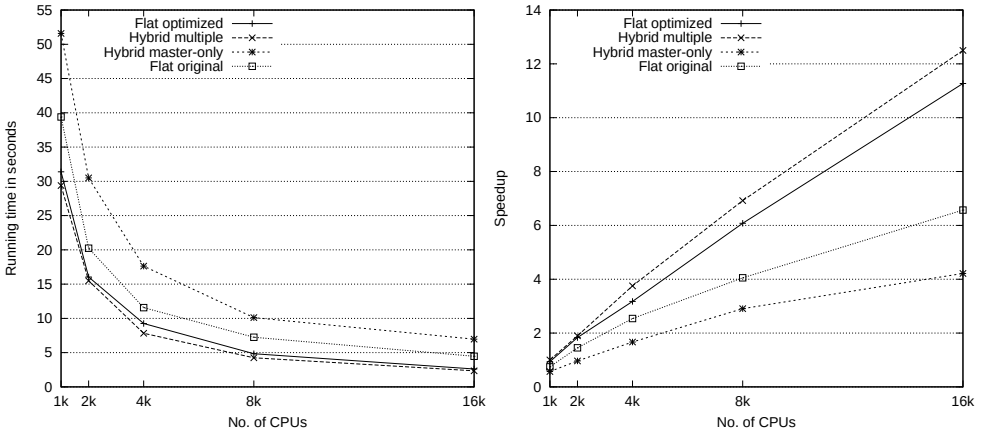


FIG. 7.4. *A scalability graph starting at 1024 CPU-cores running the stencil operation. In the left graph the running time of every approach is shown and in the right graph every approach is compared against the fastest approach on 1024 CPU-cores namely the Hybrid-multiple. All jobs consists of 2816 real-space grids all size of* 192^3*, and the best batch-size has been found for every number of CPU-cores.*

The primary improvements are obtained through the introduction of asynchronous communication which, even in a well balanced system such as the Blue Gene, efficiently improves processor utilization. Furthermore, two hybrid programming approaches have been explored: the hybrid multiple and the master-only approach.

The hybrid programming approach, in which inter-node communication is handled individually by every thread, has shown a positive impact on the performance. By allowing every thread to handle its own inter-node communication, the overhead for thread synchronization remains constant and the application becomes faster than the non-hybrid version.

On the other hand, the alternative hybrid programming approach, in which one thread handles the inter-node communication on behalf of all threads in the process, cannot compete with the non-hybrid version. That is explained by the overhead that is introduced by thread synchronization which grows proportional to the number of grids in the computation.

When comparing our fastest implementation compared to the original implementation, the hybrid programming approach combined with the latency-hiding techniques is 94% faster at 16384 CPU-cores. Translated into utilization this means that CPU utilization grows from 36% to 70%.

While latency-hiding is the primary factor for the improvement we observe, the hybrid implementation is still 10% faster than the non-hybrid approach.

8.1. Further Work. Overall we are satisfied with the performance of the new implementation of the stencil operation, still a lot of work remains if the entire GPAW computation should utilize latency-hiding and hybrid programming. It may not be worth the hard work that is needed to rewrite most of GPAW.

Acknowledgments. The authors would like to thank The Danish Agency for Science, Technology and Innovation and the GPAW team at the Technical University of Denmark in particular Jens J. Mortensen and Marcin Dulak. Furthermore we would like to thank Argonne National Laboratory for giving us access to the Blue Gene/P.

REFERENCES

[1] J. Backus, *Can programming be liberated from the von neumann style?: A functional style and its algebra of programs*, Communications of the ACM, 16 (1978), pp. 613–641.

[2] P. E. Blochl, *Projector augmented-wave method*, Phys. Rev. B, 50 (1994), pp. 17953–17979.

[3] F. Cappello and D. Etiemble, *Mpi versus mpi+openmp on the ibm sp for the nas benchmarks*, SC Conference, 0 (2000), p. 12.

[4] W. Gropp, S. Huss-Lederman, A. Limsdaine, E. Lusk, W. Saphir, and M. Snir, *The Complete Reference: Volume 2, the MPI-2 Extensions*, MIT Press, 1998.

[5] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI Portable Parallel Programming with the Message Passing Interface*, The MIT Press, 1994.

[6] D. S. Henty, *Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling*, Supercomputing, ACM/IEEE 2000 Conference, (2000), pp. 10–10.

[7] M. Hipp and W. Rosenstiel, *Parallel Hybrid Particle Simulations Using MPI and OpenMP*, Springer-Verlag Berlin Heidelberg, 2004, pp. 189–197.

[8] J. J. Mortensen, L. B. Hansen, and K. W. Jacobsen, *Real-space grid implementation of the projector augmented wave method*, Physical Review B, 71 (2005), p. 035109.

[9] I. B. G. TEAM, *Overview of the ibm blue gene/p project*, IBM Journal of Research and Development, 52 (2008).

[10] B. Vinter and J. M. Bjørndalen, *A comparison of three mpi implementations*, in Communicating Process Architectures 2004, I. R. East, D. Duce, M. Green, J. M. R. Martin, and P. H. Welch, eds., 2004, pp. 127–136.

A.3 Numerical Python for scalable architectures

Mads Ruben Burgdorff Kristensen and Brian Vinter. Numerical Python for scalable architectures
In Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model (PGAS '10). ACM, New York, NY, USA

Numerical Python for Scalable Architectures

Mads Ruben Burgdorff Kristensen
Brian Vinter

eScience Centre
University of Copenhagen
Denmark
madsbk@diku.dk/vinter@diku.dk

Abstract

In this paper, we introduce DistNumPy, a library for doing numerical computation in Python that targets scalable distributed memory architectures. DistNumPy extends the NumPy module[15], which is popular for scientific programming. Replacing NumPy with DistNumPy enables the user to write sequential Python programs that seamlessly utilize distributed memory architectures. This feature is obtained by introducing a new backend for NumPy arrays, which distribute data amongst the nodes in a distributed memory multiprocessor. All operations on this new array will seek to utilize all available processors. The array itself is distributed between multiple processors in order to support larger arrays than a single node can hold in memory.

We perform three experiments of sequential Python programs running on an Ethernet based cluster of SMP-nodes with a total of 64 CPU-cores. The results show an 88% CPU utilization when running a Monte Carlo simulation, 63% CPU utilization on an N-body simulation and a more modest 50% on a Jacobi solver. The primary limitation in CPU utilization is identified as SMP limitations and not the distribution aspect. Based on the experiments we find that it is possible to obtain significant speedup from using our new array-backend without changing the original Python code.

Keywords NumPy, Productivity, Parallel language

1. Introduction

In many scientific and engineering areas, there is a need to solve numerical problems. Researchers and engineers behind these applications often prefer a high level programming language to implement new algorithms. Of particular interest are languages that support a broad range of high-level operations directly on vectors and matrices. Also of interest is the possibility to get immediate feedback when experimenting with an application. The programming language Python combined with the numerical library NumPy[15] supports all these features and has become a popular numerical framework amongst researchers.

The idea in NumPy is to provide a numerical extension to the Python language. NumPy provides not only an API to standardized numerical solvers, but a possibility to develop new numerical solvers that are both implemented and efficiently executed in Python, much like the idea behind the MATLAB[8] framework.

NumPy is mostly implemented in C and introduces a flexible N-dimensional array object that supports a broad range of numerical operations. The performance of NumPy is significantly increased when using array-operations instead of scalar-operations on this new array.

Parallel execution is supported by a limited set of NumPy functions, but only in a shared memory environment. However, many scientific computations are executed on large distributed memory machines because of the computation and memory requirements of the applications. In such cases, the communication between processors has to be implemented by the programmer explicitly. The result is a significant difference between the sequential program and the parallelized program. DistNumPy eliminates this difference by introducing a distributed version of the N-dimensional array object. All operations on such distributed arrays will utilize all available processors and the array itself is distributed between multiple processors, which makes it possible to expand the size of the array to the aggregated available memory.

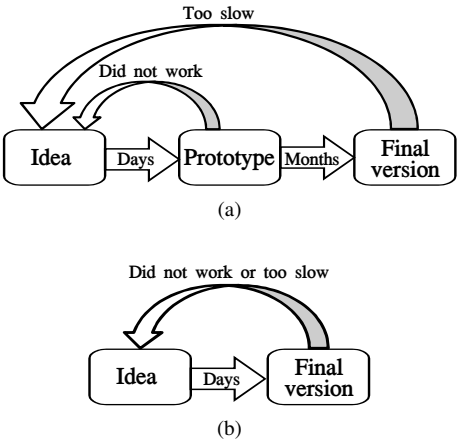


Figure 1. Development workflow. (a) is a typical workflow that involves two languages: one for the prototype and one for the final version. In (b) only one language is used in the workflow.

1.1 Motivation

Solutions to numerical problems often consist of two implementations: a prototype and a final version. The algorithm is developed and implemented in a prototype by which the correctness of the algorithm can be verified. Typical many iterations of development

PGAS'10 Oct 12–15, 2010, New York.

are required to obtain a correct prototype, thus for this purpose a high productivity language is used, most often MATLAB. However, when the correct algorithm is finished the performance of the implementation becomes essential for doing research with the algorithm. This performance requirement presents a problem for the researcher since highly optimized code requires a fairly low-level programming language such as C/C++ or Fortran. The final version will therefore typical be a reimplementation of the prototype, which involves both changing the programming language and parallelizing the implementation (Fig. 1a).

The overall target of DistNumPy is to provide a high productivity tool that meets both the need for a high productivity tool that allows researcher to move from idea to prototype in a short time, and the need for a high performance solution that will eliminate the need for a costly and risky reimplementation (Fig. 1b). It should be possible to develop and implement an algorithm using a simple notebook and then effortlessly execute the implementation on a cluster of computers while utilizing all available CPUs.

1.2 Target architectures

NumPy supports a long range of architectures from the widespread x86 to the specialized Blue Gene architecture. However, NumPy is incapable of utilizing distributed memory architectures like Blue Gene supercomputers or clusters of x86 machines. The target of DistNumPy is to close this gap and fully support and utilize distributed memory architectures.

1.3 Related work

Libraries and programming languages that support parallelization on distributed memory architectures is a well known concept. The existing tools either seek to provide optimal performance in parallel applications or, like DistNumPy, seek to ease the task of writing parallel applications.

The library ScaLAPACK[2] is a parallel version of the linear algebra library LAPACK[1]. It introduces efficient parallel operations on distributed matrices and vectors. To use ScaLAPACK, an application must be programmed using MPI[7] and it is the responsibility of the programmer to ensure that the allocation of matrices and vectors comply with the distribution layout ScaLAPACK specifies.

Another library, Global Arrays[13], introduces a distributed data object (global array), which makes the data distribution transparent to the user. It also supports efficient parallel operations and provides a higher level of abstraction than ScaLAPACK. However, the programmer must still explicitly coordinate the multiple processes that are involved in the computation. The programmer must specify which region of a global array is relevant for a given process.

Both ScaLAPACK and Global Arrays may be used from within Python and can even be used in combination with NumPy, but it is only possible to use NumPy locally and not with distributed operations. A more closely integrated Python project IPython[16] supports parallelized NumPy operations. IPython introduces a distributed NumPy array much like the distributed array that is introduced in this paper. Still, the user-application must use the MPI framework and the user has to differentiate between the running MPI-processes.

Co-Array Fortran[14] is a small language extension of Fortran-95 for parallel processing on Distributed Memory Machines. It introduce a Partitioned Global Address Space (PGAS) by extending Fortran arrays with a *co-array* dimension. Each process can access remote instances of an array by indexing into the co-array dimensions. A similar PGAS extension called Unified Parallel C (UPC)[3] extent the C language with a distributed array declaration. Both languages provide a high abstraction level, but users still program with the SPMD model in mind, writing code with the understanding that multiple instances of it will be executing cooperatively.

A higher level of abstraction is found in projects where the execution, seen from the perspective of the user, is represented as a sequential algorithm. The High Performance Fortran (HPF)[12] programming languages provide such an abstraction level. However, HPF requires the user to specify parallelizable regions in the code and which data distribution scheme the runtime should use.

The Simple Parallel R INTerface (SPRINT)[9] is a parallel framework for the programming language R. The abstraction level in SPRINT is similar to DistNumPy in the sense that the distribution and parallelization is completely transparent to the user.

2. NumPy

Python has become a popular language for high performance computing even though the performance of Python programs is much lower than that of compiled languages. The growing popularity is because Python is used as the coordinating language while the compute intensive tasks are implemented in a high performance language.

NumPy[15] is a library for numerical operations in Python which is implemented in the C programming language. NumPy provides the programmer with an N-dimensional array object and a whole range of supported array operations. By using the array operations, NumPy takes advantage of the performance of C while retaining the high abstraction level of Python. However, this also means that no performance improvement is obtained otherwise e.g. using a Python loop to traverse a NumPy array does not result in any performance gain.

2.1 Interfaces

The primary interface in NumPy is a Python interface and it is possible to use NumPy exclusively from Python. NumPy also provides a C interface in which it is possible to access the same functionality as in the Python interface. Additionally, the C interface also allows programmers to access low level data structures like pointers to array data and thereby provides the possibility to implement arbitrary array operations efficiently in C. The two interfaces may be used interchangeably through the Python program.

2.2 Universal functions

An important mechanism in NumPy is a concept called Universal function. A universal function (ufunc) is a function that operates on all elements in an array independently. That is, a ufunc is a vectorized wrapper for a function that takes a fixed number of scalar inputs and produces a fixed number of scalar outputs. Using ufunc can result in a significant performance boost compared to native Python because the computation-loop is executed in C.

2.2.1 Function broadcasting

To make ufunc more flexible it is possible to use arrays with different number of dimensions. To utilize this feature the size of the dimensions must either be identical or have the length one. When the ufunc is applied, all dimensions with a size of one will be *broadcasted* in the NumPy terminology. That is, the array will be duplicated along the *broadcasted* dimension (Fig. 2).

It is possible to implement many array operations efficiently in Python by combining NumPy's ufunc with more traditional numerical functions like matrix multiplication, factorization etc.

2.3 Basic Linear Algebra Subprograms

NumPy makes use of the numerical library Basic Linear Algebra Subprograms (BLAS) [11]. A highly optimized BLAS implementation exists for almost all HPC platforms and NumPy exploits

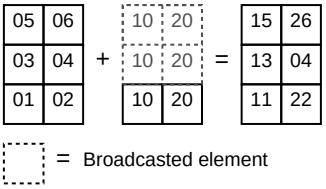


Figure 2. Universal function broadcasting. The ufunc `addition` is applied on a 3x2 array and a 1x2 array. The first dimension of the 1x2 array is broadcasted to the size of the first dimension of the 3x2 array. The result is a 3x2 array in which the two arrays are added together in an element-by-element fashion.

this when possible. Operations on vector-vector, matrix-vector and matrix-matrix (BLAS level 1, 2 and 3 respectively) all use BLAS in NumPy.

3. DistNumPy

DistNumPy is a new version of NumPy that parallelizes array operations in a manner completely transparent to the user – from the perspective of the user, the difference between NumPy and DistNumPy is minimal. DistNumPy can use multiple processors through the communication library Message Passing Interface (MPI)[7]. However, we have chosen not to follow the standard MPI approach in which the same user-program is executed on all MPI-processes. This is because the standard MPI approach requires the user to differentiate between the MPI-processes, e.g. sequential areas in the user-program must be guarded with a branch based on the MPI-rank of the process. In DistNumPy MPI communication must be fully transparent and the user needs no knowledge of MPI or any parallel programming model. However, the user is required to use the array operations in DistNumPy to obtain any kind of speedup. We think this is a reasonable requirement since it is also required by NumPy.

The only difference in the API of NumPy and DistNumPy is the array creation routines. DistNumPy allow both distributed and non-distributed arrays to co-exist thus the user must specify, as an optional parameter, if the array should be distributed. The following illustrates the only difference between the creation of a standard array and a distributed array:

```
#Non-Distributed
A = numpy.array([1,2,3])
#Distributed
B = numpy.array([1,2,3], dist=True)
```

3.1 Interfaces

There are two programming interfaces in NumPy – one in Python and one in C. We aim to support the complete Python interface and a great subset of the C interface. However, the part of the C interface that involves direct access to low level data structures will not be supported. It is not feasible to return a C-pointer that represents the elements in a distributed array.

3.2 Data layout

Two-Dimensional Block Cyclic Distribution is a very popular distribution scheme and it is used in numerical libraries like ScaLAPACK[2] and LINPACK[5]. It supports matrices and vectors and has a good load balance in numerical problems that have a diagonal computation workflow e.g. Gaussian elimination. The distribution scheme works by arranging all MPI-processes in a two dimensional grid and then distributing data-blocks in a round-robin fashion either along one or both grid dimensions (Fig. 3); the result is a well-balanced distribution.

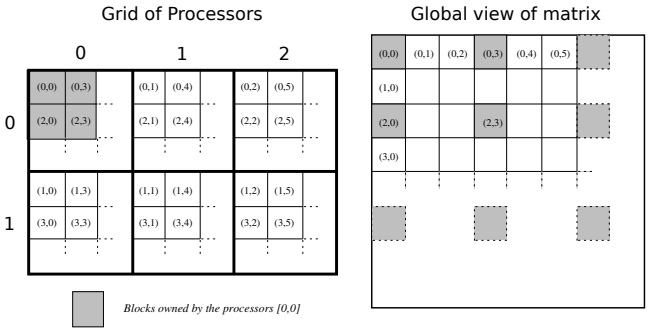


Figure 3. The Two-Dimensional Block Cyclic Distribution of a matrix on a 2 x 3 grid of processors.

NumPy is not limited to matrices and vectors as it supports arrays with an arbitrary number of dimensions. DistNumPy therefore use a more generalized N-Dimensional Block Cyclic Distribution inspired by High Performance Fortran[12], which supports an arbitrary number of dimensions. Instead of using a fixed process grid, we have a process grid for every number of dimensions. This works well when operating on arrays with the same number of dimensions but causes problems otherwise. For instance in a matrix-vector multiplication the two arrays are distributed on different process grid and may therefore require more communication. ScaLAPACK solves the problem by distributing vectors on two-dimensional process grids instead of one-dimensional process grids, but this will result in vector operations that cannot utilize all available processors. An alternative solution is to redistribute the data between a series of identically leveled BLAS operations using a fast runtime redistribution algorithm like [18] demonstrates.

3.3 Operation dispatching

The MPI-process hierarchy in DistNumPy has one MPI-process (master) placed above the others (slaves). All MPI-processes run the Python interpreter but only the master executes the user-program, the slaves will block at the `import numpy` statement.

The following describes the flow of the dispatching:

1. The master is the dispatcher and will, when the user applies a python command on a distributed array, compose a message with meta-data describing the command.
2. The message is then broadcasted from the master to the slaves with a blocking MPI-broadcast. It is important to note that the message only contains meta-data and not any actual array data.
3. After the broadcast, all MPI-processes will apply the command on the sub-array they own and exchange array elements as required (Point-to-Point communication).
4. When the command is completed, the slaves will wait for the next command from the master and the master will return to the user's python program. The master will return even though some slaves may still be working on the command, synchronization is therefore required before the next command broadcast.

3.4 Views

In NumPy an array does not necessarily represent a complete contiguous block of memory. An array is allowed to represent a subpart of another array i.e. it is possible to have a hierarchy of arrays where only one array represent a complete contiguous block of memory and the other arrays represent a subpart of that memory.

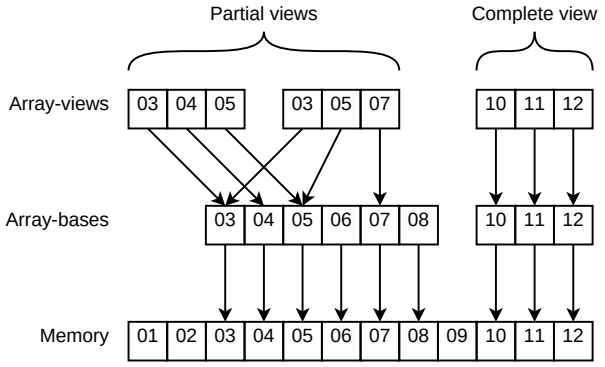


Figure 4. Reference hierarchy between the two array data structures and the main memory. Only the three array-views at top of the hierarchy are visible from the perspective of the user.

Inspired by NumPy, DistNumPy implements an array hierarchy where distributed arrays are represented by the following two data structures.

- **Array-base** is the base of an array and has direct access to the content of the array in main memory. An array-base is created with all related meta-data when the user allocates a new distributed array, but the user will never access the array directly through the array-base. The array-base always describes the whole array and its meta-data such as array size and data type are constant.
- **Array-view** is a view of an array-base. The view can represent the whole array-base or only a sub-part of the array-base. An array-view can even represent a non-contiguous sub-part of the array-base. An array-view contains its own meta-data that describe which part of the array-base is visible and it can add non-existing 1-length dimensions to the array-base. The array-view is manipulated directly by the user and from the users perspective the array-view is the array.

Array-views are not allowed to refer to each other, which means that the hierarchy is flat with only two levels: array-base below array-view. However, multiple array-views are allowed to refer to the same array-base. This hierarchy is illustrated in Figure 4.

3.5 Optimization hierarchy

It is a significant performance challenge to support array-views that represent a non-contiguous sub-part of an array-base. The difficulty lies in the process of packing communication and computation together in large blocks, which is very expensive when operating on non-contiguous data. To overcome this problem we introduce a hierarchy of implementations all optimized for specific operation scenarios. When an operation is applied a search through the hierarchy determines the most optimized implementation for that particular operation. All operations have its own hierarchy some with more levels than others, but at the bottom of the hierarchy all operations have an implementation that can handle all scenarios.

3.6 Parallel BLAS

As previously mentioned NumPy supports BLAS operations on vectors and matrices. DistNumPy therefore implements a parallel version of BLAS inspired by PBLAS from the ScaLAPACK library. Since DistNumPy uses the same data-layout as ScaLAPACK, it would be straightforward to use PBLAS for all parallel BLAS operations. However, to simplify the installation and maintenance of DistNumPy we have chosen to implement our own parallel version of BLAS. We use SUMMA[6] for matrix multiplication, which enable us to use the already available BLAS library locally on the MPI-processes. SUMMA is only applicable on complete array-views and we therefore use a straightforward implementation that computes one element at a time if partial array-views are involved in the computation.

```
1 from numpy import *
2 (x, y) = (empty([S], dist=True), \
3           empty([S], dist=True))
4 (x, y) = (random(x), random(y))
5 (x, y) = (square(x), square(y))
6 z = (x + y) < 1
7 print add.reduce(z) * 4.0 / S #The result
```

Figure 5. Computing Pi using Monte Carlo simulation. `S` is the number of samples used. We have defined a new ufunc (`ufunc_random`) to make sure that we use an identical random number generator in all benchmarks. The ufunc uses "`rand()/(double)RAND_MAX`" from the ANSI C standard library (`stdlib.h`) to generate numbers.

3.7 Universal function

In DistNumPy, the implementation of ufunc uses three different scenarios.

1. In the simplest scenario we have a perfect match between all elements in the array-views and applying an ufunc does not require any communication between MPI-processes. The scenario is applicable when the ufunc is applied on complete array-views with identical shapes.
2. In the second scenario the array-views must represent a continuous part of the underlying array-base. The computation is parallelized by the data distribution of the output array and data blocks from the input arrays are fetched when needed. We use non-blocking one-side communication (`MPI_Get`) when fetching data blocks, which makes it possible to compute one block while fetching the next block (double buffering).
3. The final scenario does not use any simplifications and works with any kind of array-view. It also uses non-blocking one-side communication but only one element at a time.

4. Examples

To evaluate DistNumPy we have implemented three Python programs that all make use of NumPy's vector-operations (ufunc). They are all optimized for a sequential execution on a single CPU and the only program change we make, when going from the original NumPy to our DistNumPy, is the array creation argument `dist`. A walkthrough of a Monte Carlo simulation is presented as an example of how DistNumPy handles Python executions.

4.1 Monte Carlo simulation

We have implemented an efficient Monte Carlo Pi simulation using NumPy's ufunc. The implementation is a translation of the Monte Carlo simulation included in the benchmark suite SciMark 2.0[17], which is written in Java. It is very simple and uses two vectors with length equal the number of samples used in the calculation. Because of the memory requirements, this drastically reduces the maximum number of samples. Combining multiple simulations will allow more samples but we will only use one simulation. The implementation is included in its full length (Fig. 5) and the following is a walkthrough of a simulation (the bullet-numbers represents line numbers):

```
1  h = zeros(shape(B), float, dist=True)
2  dmax = 1.0
3  AD = A.diagonal()
4  while(dmax > tol):
5    hnew = h + (B - add.reduce(A * h, 1)) /
         AD
6    tmp = absolute((h - hnew) / h)
7    dmax = maximum.reduce(tmp)
8    h = hnew
9  print h #The result
```

Figure 6. Iteratively Jacobi solver for matrix `A` with solution vector `B` both are distributed arrays. The `import` statement and the creation of `A` and `B` is not included here. `tol` is the maximum tolerated value of the diagonal-element with the highest value (`dmax`).

1: All MPI-processes interpret the `import` statement and initiate DistNumPy. Besides calling `MPI_Init()` the initialization is identical to the original NumPy but instead of returning from the import statement, the slaves, MPI-processes with rank greater than zero, listen for a command message from the master, the MPI-process with rank zero.

2-3: The master sends two `CREATE_ARRAY` messages to all slaves. The two messages contain an array shape and unique identifier (UID), which in this case identifies `x` and `y`, respectively. All MPI-processes allocate memory for the arrays and stores the array information.

4: The master sends two `UFUNC` messages to all slaves. Each message contains a UID and a function name `ufunc_random`. All MPI-processes apply the function on the array with the specified UID. A pointer to the function is found by calling `PyObject_Get AttrString` with the function name. It is thereby possible to support all ufuncs from NumPy.

5: Again the master sends two `UFUNC` messages to all slaves but this time with function name `square`.

6: The master sends a `UFUNC` messages with function name `add` followed by a `UFUNC` messages with function name `less_than`. The scalar `1` is also in the message.

7: The master sends a `UFUNC_REDUCE` messages with function name `add`. The result is a scalar, which is not distributed, and the master therefore solely computes the remainder of the computation and print the result. When the master is done a `SHUTDOWN` message is sent to the slaves and the slaves call `exit(0)`.

4.2 Jacobi method

The Jacobi method is an algorithm for determining the solutions of a system of linear equations. It is an iterative method that uses a spitting scheme to approximate the result.

Our implementation uses ufunc operations in a while-loop until it converges. Most of the implementation is included here(Fig. 6).

4.3 Newtonian N-body simulation

A Newtonian N-body simulation is one that studies how bodies, represented by a mass, a location, and a velocity, move in space according to the laws of Newtonian physics. We use a straightforward algorithm computing all body-body interactions. The NumPy implementation is a direct translation of a MATLAB program[4]. The working loop of the two implementations take up 19 lines in Python and 22 lines in MATLAB thus it is too big to include here. However, the implementation is straightforward and use universal functions and matrix multiplications.

Table 1. Hardware specifications

CPU	Core 2 Quad	Nehalem
CPU Frequency	2.26 GHz	2.66 GHz
CPU per node	1	2
Cores per CPU	4	4
Memory per node	8 GB @ 6.5 GB/s	24 GB @ 25.6 GB/s
Number of nodes	8	8
Network	Gigabit Ethernet	Gigabit Ethernet

5. Experiments

In this section, we will conduct performance benchmarks on DistNumPy and NumPy[1]. We will benchmark the three Python programs presented in Section 4. All benchmarks are executed on two different Linux clusters – an Intel Core 2 Quad cluster and an Intel Nehalem cluster. Both clusters consist of processors with four CPU-cores, but the number of processors per node differs. Intel Core 2 Quad cluster has one CPU per node whereas the Intel Nehalem cluster has two CPUs per node. The interconnect is Gigabit Ethernet in both clusters. (Table 1).

Our experiments consist of a speedup benchmark, which we define as an execution time comparison between a sequential execution with NumPy and a parallelized execution with DistNumPy while the input is identical. Strong-scaling is used in all benchmarks and the input size is therefore constant.

5.1 Monte Carlo simulation

A Distributed Monte Carlo simulation is embarrassingly parallel and requires a minimum of communication. This is also the case when using DistNumPy because ufuncs are only applied on identically shaped arrays and it is therefore the simplest ufunc scenario. Additionally, the implementation is CPU-intensive because a complex ufunc is used as random number generator.

The result of the speedup benchmark is illustrated in Figure 7. We see a close to linear speedup for the Nehalem cluster – a CPU utilization of 88% is achieved on 64 CPU-cores. The penalty of using multiple CPU-cores per node is noticeable on the Core 2 architecture – a CPU utilization of 68% is achieved on 32 CPU-cores.

5.2 Jacobi method

The dominating part of the Jacobi method, performance-wise, is the element-by-element multiplication of `A` and `h` (Fig. 6 line 5). It consists of $O(n^2)$ operations where as all the other operations only consist $O(n)$ operations. Since scalar-multiplication is a very simple operation, the dominating ufunc in the implementation is memory-intensive.

The result of the speedup benchmark is illustrated in Figure 8. We see a good speedup with 8 CPU-cores and to some degree also with 16 Nehalem CPU-cores. However, the CPU utilization when using more than 16 CPU-cores is very poor. The problem is memory bandwidth – since we use multiple CPU-cores per node when using more than 8 CPU-cores, the aggregated memory bandwidth of the Core 2 cluster does only increase up to 8 CPU-cores. The Nehalem cluster is a bit better because it has two memory buses per node, but using more than 16 CPU-cores will not increase the aggregated memory bandwidth.

5.2.1 Profiling of the Jacobi implementation

To investigate the memory bandwidth limitation observed in the Jacobi execution we have profiled the execution by measuring the time spend on computation and communication (Fig. 9). As ex-

[1] NumPy version 1.3.0

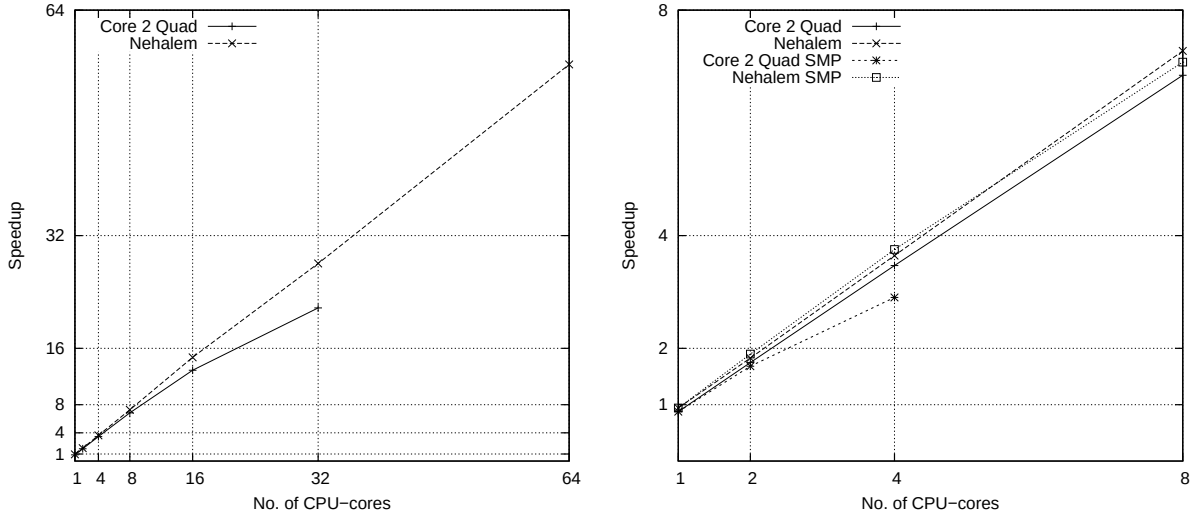


Figure 7. Speedup of the Monte Carlo simulation. In graph (a) the two architectures uses a minimum number of CPU-cores per node. Added in graph (b) is the result of using multiple CPU-cores on a single node (SMP).

pected the result shows that the percentages used with communication increases when the number of CPU-cores increases. Furthermore, a noteworthy observation is the almost identical communication overhead at eight CPU-cores and sixteen CPU-cores. This is due the change from a single CPU-core per node to multiple CPU-cores per node. At sixteen CPU-cores half of the communication is performed through the use of shared memory, which means that the communication, just like the computation, is bound by the limited memory bandwidth.

5.3 Newtonian N-body simulation

The result of the speedup benchmark is illustrated in Figure 10. Compared to the Jacobi method we see a similar speedup and CPU utilization. This is expected because the dominating operations are also simple ufuncs. Even though there are some matrix-multiplications, which have a great scalability, it is not enough to significantly boost the overall scalability.

5.4 Alternative programming language

DistNumPy introduces a performance overhead compared to a lower-level programming language such as C/C++ or Fortran. To investigate this overhead we have implemented the Jacobi benchmark in C. The implementation uses the same sequential algorithm as the NumPy and DistNumPy implementations.

Executions on both architectures show that DistNumPy and NumPy is roughly 50% slower than the C implementation when executing the Jacobi method on one CPU-core. This is in rough runtime numbers: 21 seconds for C, 31 seconds for NumPy and 32 seconds for DistNumPy.

Obviously highly hand-optimized implementations have a clear performance advantages over DistNumPy. For instance by the use of a highly optimized implementation in C [10] demonstrates extreme scalability of a similar Jacobi computation – an execution by 16384 CPU-cores achieves a CPU utilization of 70% on a Blue Gene/P architecture.

5.5 Summary

The benchmarks clearly show that DistNumPy has both good performance and scalability when execution is not bound by the memory bandwidth, which is evident from looking at the CPU utilization when only one CPU-core per node is used. As expected the scalability of the Monte Carlo simulation is better than the Jacobi and the N-body computation because of the reduced communication requirements and more CPU-intensive ufunc operation.

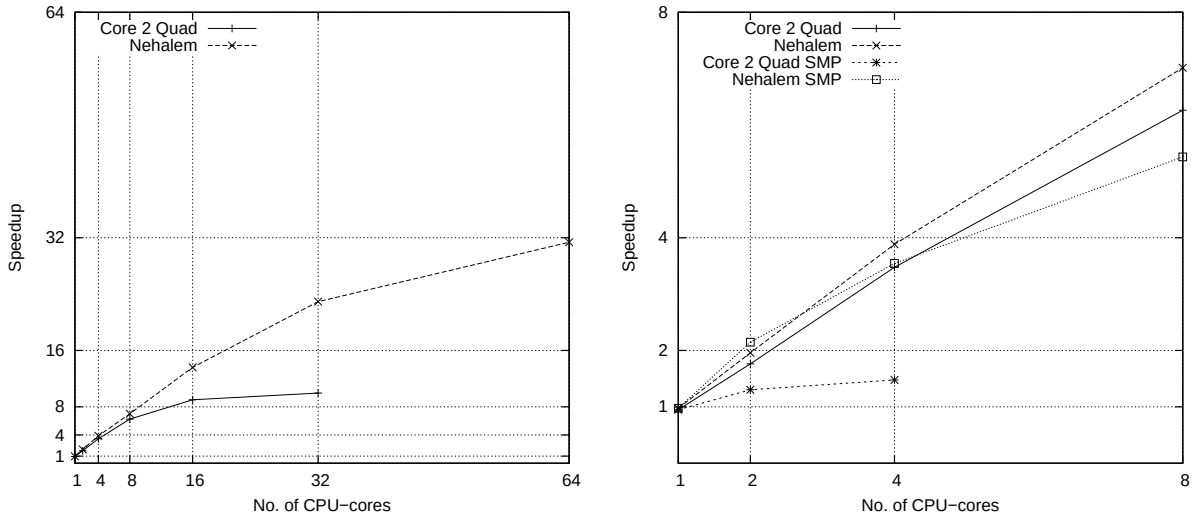
The scalability of the Jacobi and the N-body computation is drastically reduced when using multiple CPU-cores per node. The problem is the complexity of the ufunc operations. As opposed to the Monte Carlo simulation, which makes use of a complex ufunc, the Jacobi and the N-body computation only use simple ufuncs e.g. add and multiplication.

As expected the performance of the C implementation is better than the DistNumPy implementation. However, by utilizing two CPU-cores it is possible to outperform the C implementation in the case of the Jacobi method. This is not a possibility in the case of the Monte Carlo simulation where the algorithm does not favor vectorization.

6. Future work

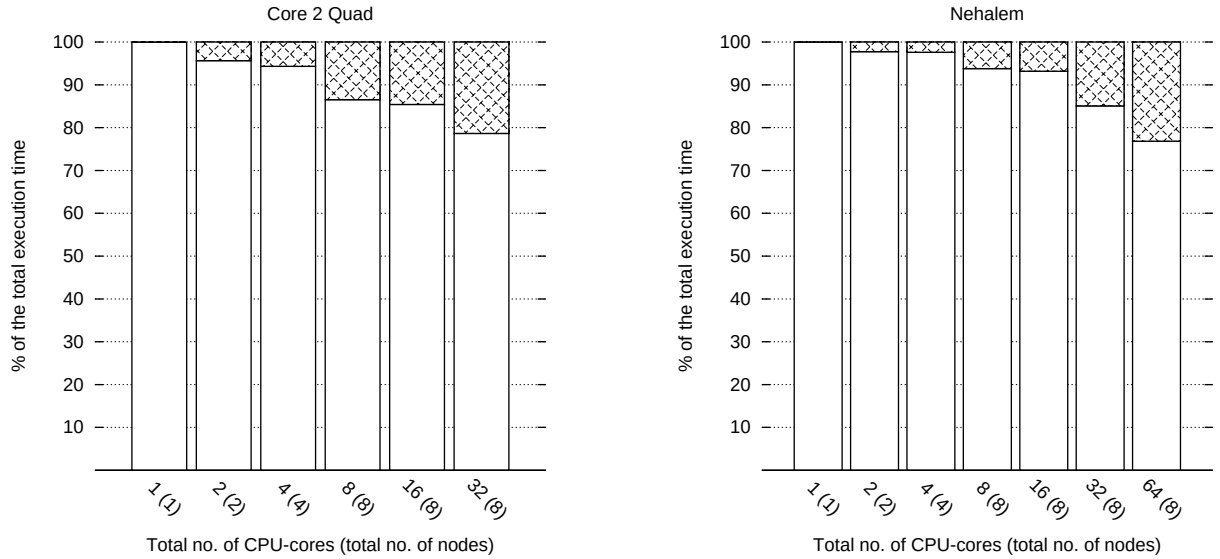
In its current state DistNumPy does not implement the NumPy interface completely. Many specialized operations like Fast Fourier transform or LU factorization is not implemented, but it is our intention to implement the complete Python interface and most of the C interface.

Other important future work include performance and scalability improvement. As showed by the benchmarks, applications that are dominated by non-complex ufuncs easily become memory bounded. One solutions is to block ufuncs, that operate on common arrays, together in one joint operation and thereby make the joint operation more CPU-intensive. If it is possible to join enough ufuncs together the application may become CPU bound rather than memory bound.



(a) (b)

Figure 8. Speedup of the Jacobi solver. In graph (a) the two architectures uses a minimum number of CPU-cores per node. Added in graph (b) is the result of using multiple CPU-cores on a single node (SMP).



(a) (b)

Figure 9. Profiling of the Jacobi experiment. The two figures illustrate the relationship between communication and computation when running on the Core 2 Quad architecture (a) and the Nehalem architecture (b). The area with the check pattern represent MPI communication and the clean area represent computation. Note that these figures relates directly to the Jacobi speedup graph (Fig 8a).

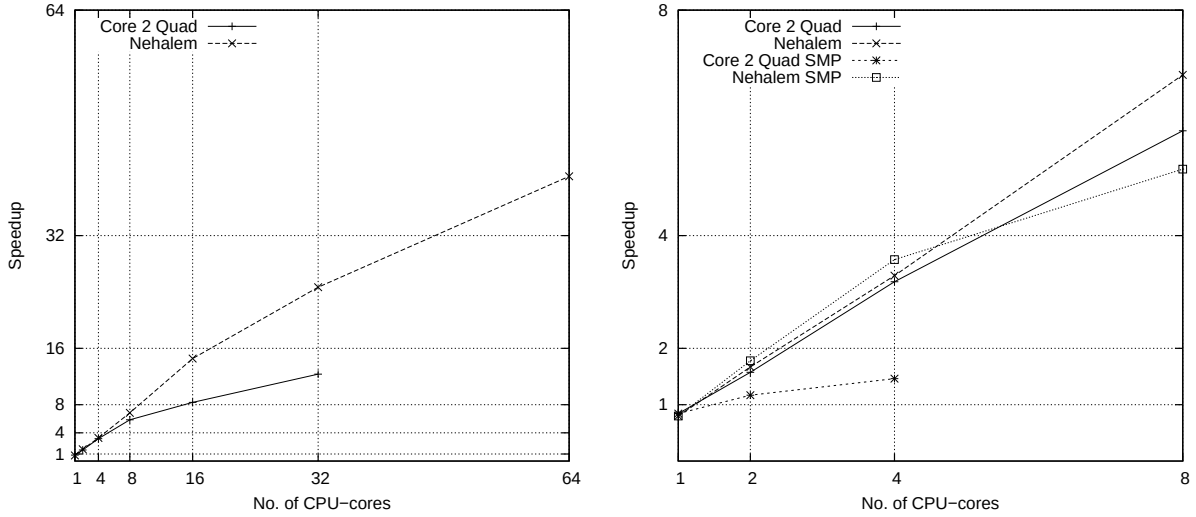


Figure 10. Speedup of Newtonian N-body simulation. In graph (a) the two architectures uses a minimum number of CPU-cores per node. Added in graph (b) is the result of using multiple CPU-cores on a single node (SMP).

7. Conclusions

In this work we have successfully shown that it is possible to implement a parallelized version of NumPy[15] that seamlessly utilize distributed memory architectures. The only API difference between NumPy and our parallelized version, DistNumPy, is an extra optional parameter in the array creation routines.

Performance measurements of three Python program, which make use of DistNumPy, show very good performance and scalability. A CPU utilization of 88% is achieved on a 64 CPU-core Nehalem cluster running a CPU-intensive Monte Carlo simulation. A more memory-intensive N-body simulation achieves a CPU utilization of 91% on 16 CPU-cores but only 63% on 64 CPU-cores. Similar a Jacobi solver achieves a CPU utilization of 85% on 16 CPU-cores and 50% on 64 CPU-cores.

To obtain good performance with NumPy the user is required to make use of array operations rather than using Python loops. DistNumPy take advantage of this fact and parallelizes array operations. Thus most efficient NumPy applications should be able to benefit from DistNumPy with the distribution parameter as the only change.

We conclude that it is possible to obtain significant speedup with DistNumPy. However, further work is needed if shared memory machines are to be fully utilized as nodes in a scalable architecture.

References

[1] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. Lapack: a portable linear algebra library for high-performance computers. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press. ISBN 0-89791-412-0.

[2] L. S. Blackford. Scalapack. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing 96 Supercomputing 96*, page 5, 1996. doi: 10.1145/369028.369038.

[3] W. W. Carlson, J. M. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to upc and language specification. Technical Report CCS-TR-99-157, Bowie, MD, May 1999.

[4] H. Casanova. N-body simulation assignment, Nov 2008. URL `http://navet.ics.hawaii.edu/~casanova/courses/ics632_fall08/projects.html`.

[5] J. Dongarra, J. Bunch, C. Moler, and G. Stewart. Linpack users' guide. *SIAM*, 1, 1979.

[6] R. A. v. d. Geijn and J. Watts. Summa: scalable universal matrix multiplication algorithm. *Concurrency - Practice and Experience*, 9 (4):255–274, 1997.

[7] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI Portable Parallel Programming with the Message Passing Interface*. The MIT Press, 1994.

[8] M. U. Guide. The mathworks. *Inc., Natick, MA*, 5, 1998.

[9] J. Hill, M. Hambley, T. Forster, M. Mewissen, T. M. Sloan, F. Scharinger, A. Trew, and P. Ghazal. Sprint: a new parallel framework for r. *BMC Bioinformatics*, 9:558, 2008. doi: 10.1186/1471-2105-9-558.

[10] M. R. B. Kristensen, H. H. Happe, and B. Vinter. Gpaw optimized for blue gene/p using hybrid programming. *Parallel and Distributed Processing Symposium, International*, 2009. doi: 10.1109/IPDPS.2009.5160936.

[11] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979. doi: 10.1145/355841.355847.

[12] D. Loveman. High performance fortran. *IEEE Parallel & Distributed Technology: Systems & Applications*, 1(1):25, 1993. doi: 10.1109/88.219857.

[13] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2), 1996. doi: 10.1007/BF00130708.

[14] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998. ISSN 1061-7264. doi: 10.1145/289918.289920.

[15] T. E. Oliphant. Python for scientific computing. *Computing in Science and Engineering*, 9:10–20, 2007. ISSN 1521-9615. doi: 10.1109/MCSE.2007.58.

[16] F. Pérez and B. E. Granger. Ipython: a system for interactive scientific computing. *Comput. Sci. Eng.*, 9(3):21–29, may 2007.

[17] R. Pozo and B. Miller. Scimark 2.0, 12 2002. URL `http://math.nist.gov/scimark2/`.

[18] L. Prylli and B. Tourancheau. Fast runtime block cyclic data redistribution on multiprocessors. *J. Parallel Distrib. Comput*, 45(1):63–72, 1997.

A.4 Managing Overlapping Data Structures for Data-Parallel Applications on Distributed Memory Architectures

Mads Ruben Burgdorff Kristensen and Brian Vinter. Managing Overlapping Data Structures for Data-Parallel Applications on Distributed Memory Architectures
GSTF International Journal on Computing (JoC), vol. 1, no. 2, pp. 145-151, 2012. ISSN: 2010-2283.

Managing Overlapping Data Structures for Data-Parallel Applications on Distributed Memory Architectures

Mads Ruben Burgdorff Kristensen and Brian Vinter

Abstract— **In this paper, we introduce a model for managing abstract data structures that map to arbitrary distributed memory architectures. It is difficult to achieve scalable performance in data-parallel applications where the programmer manipulates abstract data structures rather than directly manipulating memory. On distributed memory architectures such abstract data-parallel operations may require communication between nodes. Therefore, the underlying system has to handle communication efficiently without any help from the user. Our data model splits data blocks into two sets -- local data and remote data -- and schedules the sub-block by availability at runtime.**

We implement the described model in DistNumPy -- a high-productivity programming library for Python. We go on to evaluate the implementation using a representative distributed memory system -- a Cray XE-6 Supercomputer -- up to 2048 cores. The benchmarking results demonstrate scalable good performance.

Index Terms—**HPC, NumPy, High-Productivity, Data-Parallel, DistNumPy**

I. Introduction

High-productivity programming languages are very popular in the computational scientific community because they enable quickly prototyping of numerical problems. Common for most high-productivity languages is high-level operation on data structures such as vectors and matrices because they increase the productivity and remove a broad range of typical errors. Two high-productivity languages, MATLAB and Python, are popular in the scientific community precisely because of a rich set of high-level vector and matrix operations.

It is possible to execute parallel applications written in a high-productivity language that make use of data parallelism without reducing the productivity[4, 11]. This is because data parallelism is ideal for high-level vector and matrix operations. Data parallelism refers to a parallel model where a single instruction is distributed between processes based on data locality. Therefore, data parallelism provides full knowledge of data distribution and parallelization to all participating processors, which makes it possible for the runtime system to execute vector operations seamlessly in parallel without further assistance from the user. Additionally, the processors need not communicate when performing data dependency analysis and scheduling optimizations at runtime. However, the downside of data parallelism is that it reduces the programmability because the user is restricted to vector operations.

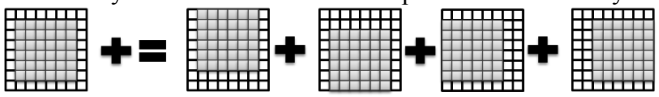


Fig.1, Matrix expression of a simple 5-point stencil computation example. See Figure 2 for the expression in MATLAB and Figure 8 for the expression in Python.

```
1 I %Number of iterations
2 A %Input & Output Matrix
3 SIZE %Symmetric Matrix Size
4 T = %Temporary array
5 i = 2:SIZE+1;%Center slice vertical
6 j = 2:SIZE+1;%Center slice horizontal
7 for n=1:I,
8     T(:) = A(i,j) + A(i+1,j) + A(i-1,j) ...
9          + A(i,j+1) + A(i,j-1);
10    A(i,j) = T;
11 end
```

Fig. 2, 5-point stencil application that uses Jacobi Iteration in a fixed number of iterations implemented in MATLAB.

When expressing algorithms through high-level vector and matrix operations, or simply array operations, the user needs a mechanism to specify a subset of an array. E.g., Figure 1and 2 illustrate how one implements a 5-point-stencil computation in MATLAB by operating on *views* of arrays. In contrast, conventional programming languages would require using tedious scalar operations with *for* loops and index arithmetic.

These array views are data structures that maps to arbitrary distributed memory and thus possible overlapping memory. In the context of this paper, we will use array views as a synonym for such abstract data structures that may refer to parts of the same underlying data.

Array views gives rise to a number of important performance challenges when combined with data parallelism where the shared data is distributed across multiple processes. The problem is that operations on views may translate into *non-aligned* distributed array operations, which are difficult to handle efficiently. We define an *aligned* distributed array operation as an operation on arrays that are distributed in a

Manuscript received February 16, 2012. Mads R. B. Kristensen and Brian Vinter is with Niels Bohr Institute, University of Copenhagen, Denmark. Email: madsbk@nbi.dk/vinter@nbi.dk.

conformable manner, i.e. the arrays use identical data distribution. A non-aligned distributed array operation is then an operation without this property.

In this paper, we will introduce a data model that provides efficient handling of overlapping data structures. We will concretize the data model by implementing efficient array views in the high-productivity language DistNumPy[11], which interprets NumPy applications as data parallel applications in a distributed memory environment. In order to achieve good scalable performance we leverage the work by [14] who introduce an efficient communication latency-hiding model.

A. Related Work

Libraries and programming languages that strive to support parallelism in a high productive manner is a well-known concept. In a perfect framework all parallelism introduced by the framework is completely hided from the user while the performance and scalability archived is optimal. However, most frameworks require the user to specify some kind of parallelism -- either explicitly by using parallel directives or implicitly by using parallel data structures.

High Performance Fortran (HPF)[12] and ZPL[3] are two well-known examples of data-parallel programming languages that supports abstract data structures. HPF is a Fortran-based data-parallel programming language that requires static compilation for distributed-memory systems[10]. To obtain good parallel performance the user must *align* arrays together to reduce communication[1]. Our data model manages computation and communication of abstract data structures at runtime, which enables on-the-fly data dependency analysis. Using our model the user will not have to *align* arrays in order to obtain good parallel performance.

Python extensions, NumPy[13] and SciPy[9], have been successfully used in scientific computing[6] because their high-level abstractions are very close to mathematical formulas and there exist a super rich set of Python packages for almost any common task. Similarly, MATLAB is very popular because of a high-level data structure abstraction support. NumPy, SciPy, and MATLAB are targeting single-node systems where as our model is targeting multi-node systems. There exists extension to MATLAB that targets multi-node systems. MATLAB*P[4] introduces data-parallelism in MATLAB with support for high-level data structure abstraction.

II. Target Data-Parallel Applications

Data-parallel applications are a class of applications that make use of data parallelism -- either explicitly handled by the programmer or implicitly handled by the programming language or library. In this work, we focus on data-parallel applications written in a high-productivity language where the programming language, scientific library, and/or runtime system handles the data parallelism seamlessly.

We target applications with the following properties:

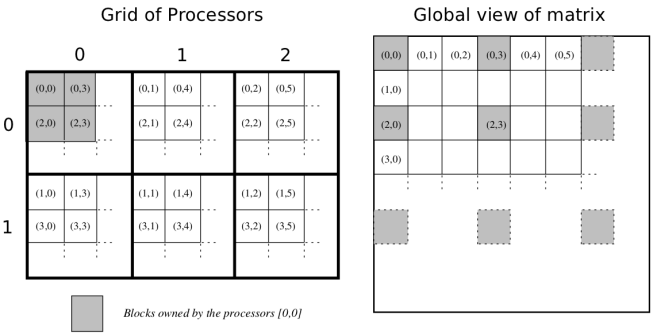


Fig. 3, The Two-Dimensional Block Cyclic Distribution of a matrix on a 2 x 3 grid of processors.

- The application uses high-level array operations instead of explicitly programmed *for* loops.
- The application uses data parallelism to execute vector/array operation in parallel.
- In order to utilize distributed memory architectures, the application distribute data evenly across process using a static distribution scheme.

The application uses data structures that maps to arbitrary distributed memory, e.g. by using data structures, such as array views, that may refer to parts of the same underlying data.

A. Data Distribution

Data parallelism is a classic approach to support distributed memory architectures. It clearly defines how data and computation is distributed across processes when combined with a static distribution scheme. Two-Dimensional Block Cyclic Distribution is a very popular distribution scheme and it is used in numerical libraries such as ScaLAPACK[2] and LINPACK[5]}. It supports matrices and vectors and has a good load balance in numerical problems that have a diagonal computation workflow e.g. Gaussian elimination. The distribution scheme works by arranging all processes in a two dimensional grid and then distributing data-blocks in a round-robin fashion either along one or both grid dimensions (Fig. 3); the result is a well-balanced distribution.

B. Array Operations

High-level array operation is relevant for all kinds of computations. Some array operations are very domain specific and other array operations are very general. Element-wise operations on arrays are an elementary part of most high-

productivity languages and libraries. It simplifies the programming because it replaces computation loops, including index arithmetic, with one single operation.

Element-wise operations take a fixed number of scalar inputs and produce a fixed number of scalar outputs. E.g., an element-wise addition takes three array-views as argument: two input arrays and one output array. For each element, the operation adds the two input arrays together and writes the result into the output array. Applying an element-wise operation on a whole array is semantically equivalent to performing the operation on each distributed array block individually. This property makes it possible to perform the distributed element-wise operation in parallel.

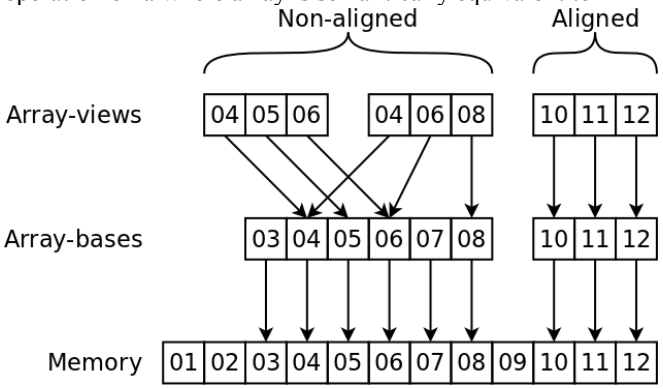


Fig. 4, Reference hierarchy between the two array data structures and the main memory. Only the three array-views at top of the hierarchy are visible from the perspective of the user.

C. Array Views

Array views are essential when expressing algorithms through high-level array operations. It makes it possible to apply an operation on a subpart of an existing array without memory copying. Conceptually, array views form a hierarchy where each array view points to an underlying ``base''. This ``base'' is then an array that maps directly to a contiguous piece of memory. We define the two terms, array-base and array-view, as follows:

- **Array-base** is the base of an array and has direct access to the content of the array in main memory. An array-base is created with all related meta-data when the user allocates a new distributed array, but the user will never access the array directly through the array-base. The array-base always describes the whole array and its meta-data such as array size and data type are constant.
- **Array-view** is a view of an array-base. The view can represent the whole array-base or only a sub-part of the array-base. An array-view can even represent a non-contiguous sub-part of the array-base. An array-view contains its own meta-data that describe which part of the array-base is visible. The array-view is manipulated directly by the user and from the users perspective the array-view is simply a normal contiguous array.

For simplicity, array-views are not allowed to refer to each other, which mean that the hierarchy is flat with only two levels: array-base below array-view. However, multiple array-views are allowed to refer to the same array-base. This hierarchy is illustrated in Figure 4.

III. Non-Aligned Array Operations

Managing overlapping data structures, aka array-view, for data-parallel applications on distributed memory architectures gives rise to a number of important performance challenges.

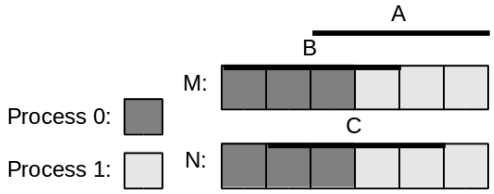


Fig. 5, The data layout of the two arrays M and N and the three array-views A, B and C in the 3-point stencil application. The arrays are distributed between two processes using a block-size of three.

The problem is that element-wise operations on array-views may translate into *non-aligned* distributed array operations, which are difficult to handle efficiently. That is, element-wise operations on array-views that does not map directly to the underlying array-base.

For example, a 3-point stencil application uses three array-views, *A*, *B* and *C*, to express a stencil. When executing on two processes the two underlying array-bases, *M* and *N*, are distributed according to Fig. 5. It is clear that *A* and *C* does not map directly to the underlying array-bases *M* and *N*. Thus, the result is a non-aligned array operation. In order to execute such an application the two processes must exchange data blocks, which mean commutation when executing on a distributed memory architecture. Therefore, an efficient data structure model that minimizes communication is vital for the parallel performance.

IV. Managing Non-Aligned Array Operations

The main contribution in this work is a model for managing non-aligned array operations efficiently. We introduce a hierarchy of data structures that makes it possible to divided non-aligned array operations into aligned blocks at runtime while minimizing the total amount of communication.

The model consists of three kinds of data blocks: base-blocks, view-blocks and sub-view-blocks, which make up a three level abstraction hierarchy (Fig. 6).

- **Base-block** is a block of an array-base and maps directly into one block of memory located on one node. The memory block is contiguous and only one process has exclusive access to the block. The base-blocks are distributed across multiple processes in a round-robin fashion according to the N-Dimensional Block Cyclic Distribution.
- **View-block** is a block of an array-view and from

the perspective of the user a view-block is a contiguous block of array elements. A view-block can span over multiple base-blocks and consequently also over multiple processes. For a process to access a whole view-block it will have to fetch data from possible remote processes and put the pieces together before accessing the block. To avoid this process, which may cause some internal memory copying, we divide view-blocks into sub-view-block.

- **Sub-view-block** is a block of data that is a part of a view-block but is located on only one process. The memory block is not necessarily contiguous but only one process has exclusive access to the block. The driving idea is that all array operation is translated into a number of sub-view-block operations.

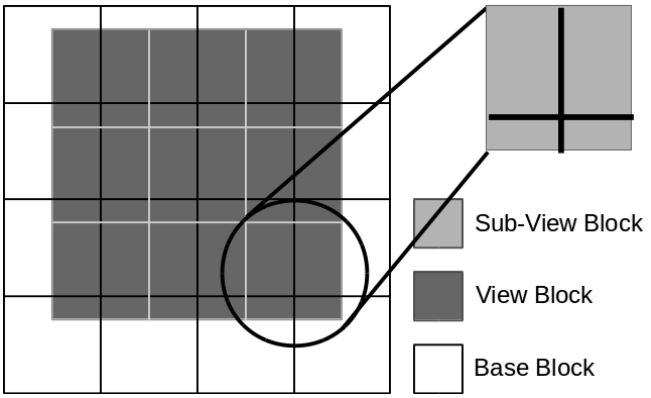


Fig. 6, An illustration of the block hierarchy that represents a 2D distributed array. The array is divided into three block-types: Base, View and Sub-View-blocks. The 16 base-blocks make up the base-array, which may be distributed between multiple processes. The nine view-blocks make up a view of the base-array and represent the elements that are visible to the user. Each view-block is furthermore divided into four sub-view-blocks, each located on a single process.

In this data model, an aligned array is an array that has a direct contiguous mapping through the block hierarchy. That is, a distributed array in which the base-blocks, view-blocks and sub-view-blocks are identical. A non-aligned array is then a distributed array without this property.

It is straightforward to parallelization aligned array operations because each view-block is identical to the underling base-block and is located on a single process. On the other hand, when operating on non-aligned arrays each view-block may be located on multiple processes. Therefore, we have to divide the computation into sub-view-blocks and even into aligned blocks of sub-view-blocks, which makes the operation more complex and introduces extra communication and computation overhead.

At the user level, an array operation operates on a number of input array-views and output array-views. It is the user's responsibility to make sure that the shape of these array-views matches each other. Since all arrays uses the same block size, this guaranties that all involved view-blocks match each other. Thus, it is possible to handle one view-block from each array at a time. In order to compute an array operation in parallel all available processes computes a view-block using the following steps:

1) The process fetches all the remote sub-view-blocks that constitute the involving input view-blocks.
2) The process aligns the sub-view-blocks by dividing them into the smaller blocks that are aligned to each other. If some output sub-view-blocks is not located on the process it will use temporary memory for the output.
3) The process applies operation on these aligned blocks.
4) The process sends temporary output sub-view-blocks back to the original locations.

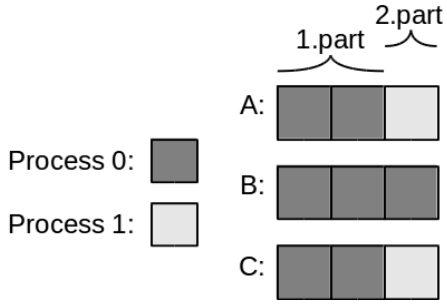


Fig. 7, The sub-view-block alignment of the first view-block in the three array-views *A*, *B* and *C* (Fig. 5).

A. References

To demonstrate how the model works we will walk through the execution of the first block in a small 3-point stencil application. Two processes are executing the stencil application with the two array-bases, *M* and *N*, using a block-size of three elements. This means that three contiguous array elements are located on each process (Fig. 5). The application uses two input array-views, *A* and *B*, and one output array-view, *C*, to compute the 3-point stencil.

In order to compute the first view-block in the three array-views, process 0 divides the computation into two parts (Fig. 7). The first part, which consists of the first two elements, needs no communication since all elements are located locally. The process can therefore apply the operation directly on the first two elements of each array.

The second part, which consists of the third element, needs communication. The two processes will transfer the third element in *A* from process 1 to process 0. Even though the third element in *C* is located remotely, no communication is need now because *C* is the output. Instead, a temporary memory location is used for the output element. The process will apply the operation when the communication the element is finished. When process 0 finishes the computation of part 2 the process transfer the third element back to process 1.

B. Latency-Hiding

It is essential to the performance of non-aligned array operations that the execution hides communication latency

behind computation. In order to accomplish this we make use of the Latency-Hiding model introduces in [14]. Using this model, we initiate non-blocking communication at the earliest time and only do computation after all communication has been initiated. Furthermore, we check for communication completion between multiple computation operations to make sure that there is progress in the communication layer. The execution flow is as follows:

1) Initiate all non-depended communication operations.
2) Check if any communication operations has been finished in a non-blocking manner and insert operations that have no dependencies into the ready queue.
3) When only computation operations are ready, execute one of them and move new operations that have no dependencies into the ready queue.
4) Go back to step one if there are unfinished operations or else terminate.

The algorithm maintains the following three invariants:

1) All ready operations are in the ready queue.
2) Computation operations are executed only when there is no communication operation in the ready queue.
3) Communication operations are checked for completion when there is no computation operation in the ready queue.

Table 1. Cray XE-6 Supercomputer

Processor	AMD Opteron 6172
Clock	2.1 GHz
Peak Performance per Core	8.4 Gflops
Cores per NUMA Domain	6
NUMA Domains per Node	4 (packaged in 2 sockets)
Total Cores per Node	24
Private L1 Data Cache	64 KB
Private L2 Data Cache	512 KB
Shared L3 Cache per Socket	12MB
Memory Bandwidth	25.6 GB/s
Memory per Node	32GB DDR3-1066 ECC
Compiler	PGI 11.3
Math Library	Cray Scientific Library 10.5
Interconnect	Gemini 3-D Torus
Peak Bandwidth (per direction)	7 GB/s
MPI	Cray MPI 5.1.4

V. Distributed Numerical Python

In order to demonstrate the efficiency of our model for managing abstract data structures, we optimize the numerical Python library Distributed Numerical Python (DistNumPy) [11] using our model. DistNumPy is a new version of NumPy[13] that parallelizes array operations in a manner completely transparent to the user -- from the perspective of the user, the difference between NumPy and DistNumPy is minimal. DistNumPy can use multiple processors through the communication library Message Passing Interface (MPI)[7]. However, DistNumPy does not use the traditional single-program multiple-data (SPMD) parallel programming model. Instead, the MPI communication in DistNumPy is fully transparent and the user needs no knowledge of MPI or any parallel programming model.

The only difference in the API of NumPy and DistNumPy is the array creation routines. DistNumPy allow both distributed and non-distributed arrays to co-exist thus the user must specify, as an optional parameter, if the array should be distributed. The following illustrates the only difference between the creation of a standard array and a distributed array:

```
#Non-Distributed
A = numpy.array([1,2,3])
#Distributed
B = numpy.array([1,2,3], dist=True)
```

The first version of DistNumPy does not support efficient non-aligned array operations. Its focus was scientific applications that uses aligned distributed array operations, such as Monte Carlo and N-body simulations. To address this shortcoming we introduce our model for managing abstract data structures efficiently. We expect good performance and scalability when combining this implementation with the latency-hiding model introduced in [14].

The implementation of DistNumPy is open-source and freely available (http://code.google.com/p/DistNumPy).

```
1 I #Number of iterations
2 A #Input & Output Matrix
3 SIZE //Symmetric Matrix Size
4 #Temporary array
5 T = empty([SIZE]*2,dtype=double,dist=True)
6 for i in xrange(I):
7   T[:] = A[1:-1, 1:-1] #Center
8   T   += A[1:-1, 0:-2] #Left
9   T   += A[1:-1, 2:  ] #Right
10  T   += A[0:-2, 1:-1] #Up
11  T   += A[2:  , 1:-1] #Down
12  A[1:-1, 1:-1] = T
```

Fig. 8, 5-point stencil application that uses Jacobi Iteration in a fixed number of iterations implement in DistNumPy.

VI. Experiments

In this section, we will evaluate the performance impact of our model for managing non-aligned array operations. We conduct all experiments on an Cray XE6 supercomputer (Table 1). The system systems consist of multi-core Non-Uniform Memory Access (NUMA) shared-memory nodes where each node has multiple NUMA domains. CPU cores within the same NUMA domain have uniform data access latency to the local memory while CPU cores of different NUMA domains would have non-uniform data access latencies. We will focus on the MPI communication overhead associated with non-aligned array operation and we will therefore only execute one MPI-process per NUMA domain.

To evaluate the performance, we will compare aligned array operations with non-aligned array operations. We use a 5-point stencil application that uses Jacobi Iteration in a fixed

number of iterations. Figure 8 is this application implemented in Python using the DistNumPy library. It expresses the 5-point stencil using five array views that are shifted one element in each direction and thereby non-aligned operations (Fig. 1). In order to benchmark the efficiency of the data structures hierarchy we introduce in this work, we compare this application with a synthetic version where all operations an aligned and do the same amount of computation. Because

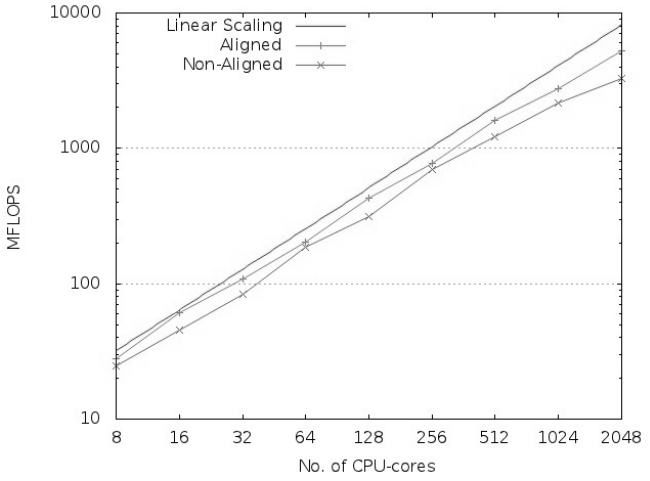


Fig. 9, Weak scaling of aligned and non-aligned array operation.

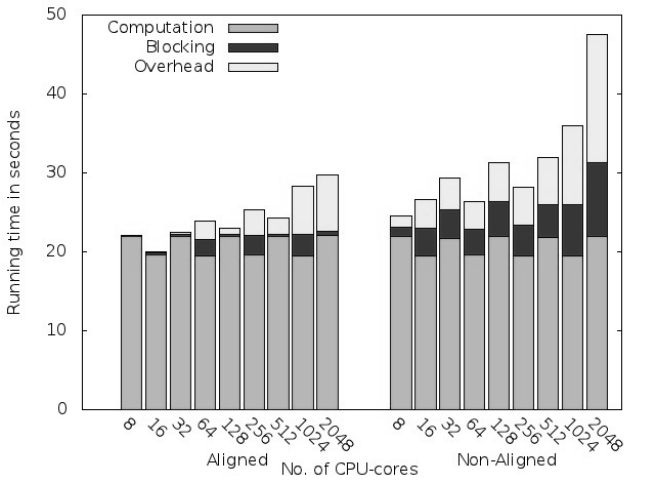


Fig. 10, Weak scaling of aligned versus non-aligned array operation.

of the exclusively use of aligned operation the synthetic version requires no communication. It should be emphasize that the synthetic version is purely for benchmark purposes and do no meaningful work.

The unfavorable computation-communication ratio in the 5-point stencil application makes it difficult to achieve good scaling performance. The asymptotic computational complexity is *O(n)* thus increasing the problem size does not improve the scaling performance significantly.

For the experiment, we calculate the FLOPS based on the floating operation counts of the ideal sequential algorithm and the measured execution times. Additionally, we compare the results with the linearly scaling performance, which we calculate by extrapolating the sequential FLOPS performance of NumPy. We use this comparison as an upper bound of the achievable scalable performance. We perform weak scaling experiments, in which the problem size is scaled with the number of CPU-cores in the executions. The experiment goes from 8 to 2048 CPU-cores where the CPU-cores and problem size doubles between each execution.

A. Results

Figure 9 shows the result of the experiment. Overall the result is very promising, we see a linear increase of performance in both the aligned and non-aligned version. The aligned version demonstrates a speedup of 1514 at 2048 CPU-cores compared to a sequential execution, which translates into a CPU utilization of 74%. The non-aligned version demonstrates a speedup of 948 at 2048 CPU-cores compared to a sequential execution, which translates into a CPU utilization of 46%.

To analyze the experiment result further we divide the execution time into three categories in Figure 10. The execution time in each category is the average timing from each process.

- **Computation** is the time used on actually computing element values. It should be fairly static through all the executions. However, variations in the data distribution may result in different execution times.
- **Blocking** is the time used on waiting for communication to finish. Each process will do as much work as possible before interring a blocking state. However, as the number of CPU-cores increases the chances that the job scheduler on the Cray system allocates distant nodes to a job also increases. Furthermore, the torus network performance may suffer from the communication traffics caused by other jobs.
- **Overhead** is the time used on handling the data structures associated with array operations. The overhead is proportional with the number of sub-view-blocks involved in the computation. Since the number of sub-view-blocks increases with the problem size, the overhead also increases. In addition, the number of sub-view-blocks increases even more when executing non-aligned operations.

As expected the blocking time is relatively small for all the aligned operation executions. Even at 2048, the blocking time is less the 2% of the total execution time. On the other hand, the blocking time for the non-aligned version is not as good. At 2048, the blocking time is 18% of the total execution time. This increase in blocking time is primarily because of an increase in communication, but also because of the MPI implementation by Cray. Currently, the Cray MPI for the Cray Gemini network has limited overlapping support for non-blocking MPI communication.

In the aligned operation version, the overhead time increases from 0.4% to 24% of the overall execution time. This overhead incensement is a direct result of the increased

problem size. In the non-aligned operation version, the overhead increases more drastically -- going from 6% to 34% of the overall execution time. This is because the non-aligned operations results in four times the number of sub-view-blocks -- one sub-view-block per direction in the stencil computation.

VII. Conclusion

The single execution flow with abstract data operations is both the main strength and weakness of data-parallel programming models: two most notorious types of parallel programming bugs, data races and deadlocks, simply do not exist in data-parallel applications because there is only one execution thread. However, flexible abstract data operations for data-parallel applications require a very efficient runtime system in order to have good scalable performance.

In this work, we have successfully shown that by splitting data blocking based on locality it is possible to efficiently managing abstract data structures that map to arbitrary distributed memory. We demonstrate scalable performance of a Jacobi Iteration application up to 2048 CPU-cores.

Acknowledgment

This research is supported by the Danish Strategic Research Council, grant #09-063770. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] S. Benkner, P. Mehrotra, J. Van Rosendale, and H. Zima. High-level management of communication schedules in HPF-like languages. In Proceedings of the 12th international conference on Supercomputing, ICS '98, pages 109–116, New York, NY, USA, 1998. Institute for Software Technology and Parallel Systems, University of Vienna. ISBN 0-89791-998-X.

[2] L. S. Blackford. ScaLAPACK. In Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing 96 Supercomputing 96, page 5, 1996.

[3] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and D. Weathersby. ZPL: A Machine Independent Programming Language for Parallel Computers. Software Engineering, 26(3):197–211, 2000.

[4] R. Choy and A. Edelman. MATLAB*P 2.0: A unified parallel MATLAB. Technical report, Massachusetts Institute of Technology, January 2003.

[5] J. Dongarra, J. Bunch, C. Moler, and G. Stewart. LINPACK users' guide. SIAM, 1, 1979.

[6] P. F. Dubois. Guest Editor's Introduction: Python: Batteries Included. Computing in Science Engineering, 9(3):7–9, may-june 2007. ISSN 1521-9615.

[7] W. Gropp, E. Lusk, and A. Skjellum. Using MPI Portable Parallel Programming with the Message Passing Interface. The MIT Press,1994.

[8] M. U. Guide. The MathWorks. Inc., Natick, MA, 5, 1998.

[9] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001.

[10] K. Kennedy, C. Koelbel, and H. Zima. The rise and fall of High Performance Fortran: an historical object lesson. In Proceedings of the third ACM SIGPLAN conference on History of programming languages, HOPL III, pages 7–1, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-7.

[11] M. R. B. Kristensen and B. Vinter. Numerical Python for Scalable Architectures. In Fourth Conference on Partitioned Global Address Space Programming Model, PGAS'10. ACM, 2010. ISBN 978-1-4503-0461-0.

[12] D. Loveman. High Performance Fortran. IEEE Parallel & Distributed Technology: Systems & Applications, 1(1):25, 1993.

[13] T. E. Oliphant. Python for Scientific Computing. Computing in Science and Engineering, 9:10–20, 2007. ISSN 1521-9615.

[14] M. Ruben Burgdorff Kristensen and B. Vinter. Managing Communication Latency-Hiding at Runtime for Parallel Programming Languages and Libraries. Arxiv Preprint arXiv:1201.3804v1, jan 2012.

Mads Ruben Burgdorff Kristensen is a PhD student at the Niels Bohr Institute, University of Copenhagen. His primary PhD study is in Supercomputing and Multi-core architectures. Current work includes seamlessly parallelism in scientific Python applications with special focus on exploiting distributed memory architectures.

Brian Vinter is Professor at the Niels Bohr Institute, University of Copenhagen. His primary research areas are Grid computing, Supercomputing and Multi-core architectures. He has done research in the field of High Performance Computer since 1994. Current research includes methods for transparent utilization of parallelism in scientific applications with special focus on exploiting distributed memory architectures such as CELL-BE and BlueGene.

A.5 Managing Communication Latency-Hiding at Runtime for Parallel Programming Languages and Libraries

Mads Ruben Burgdorff Kristensen and Brian Vinter. Managing Communication Latency-Hiding at Runtime for Parallel Programming Languages and Libraries
In Proceedings of the 2012 IEEE International Conference on High Performance Computing and Communications (HPCC'12). IEEE.

Managing Communication Latency-Hiding at Runtime for Parallel Programming Languages and Libraries

Mads Ruben Burgdorff Kristensen
Niels Bohr Institute
University of Copenhagen
Denmark
madsbk@nbi.dk

Brian Vinter
Niels Bohr Institute
University of Copenhagen
Denmark
vinter@nbi.dk

***Abstract*—This work introduces a runtime model for managing communication with support for latency-hiding. The model enables non-computer science researchers to exploit communication latency-hiding techniques seamlessly. For compiled languages, it is often possible to create efficient schedules for communication, but this is not the case for interpreted languages. By maintaining data dependencies between scheduled operations, it is possible to aggressively initiate communication and lazily evaluate tasks to allow maximal time for the communication to finish before entering a wait state. We implement a heuristic of this model in DistNumPy, an auto-parallelizing version of numerical Python that allows sequential NumPy programs to run on distributed memory architectures. Furthermore, we present performance comparisons for six benchmarks with and without automatic latency-hiding. The results shows that our model reduces the time spent on waiting for communication as much as 27 times, from a maximum of 54% to only 2% of the total execution time, in a stencil application.**

I. Introduction

There are many ways to categorize scientific applications – terms including scalability, communication pattern, IO and so forth. In the following, we wish to differentiate between large maintained codes, often commercial or belonging to a community, and smaller, less organized, codes that are used by individual researchers or in a small research group. The large codes are often fairly static and each version of the code can be expected to be run many times by many users, and thus justifying a large investment in writing the code. The small development codes on the other hand, change frequently and may only be run a few times after each change, usually only by the one user who made the changes.

The consequence of these two patterns is that the large codes may be written in a compiled language with explicit message-passing. While the small codes have an inherent need to be written in a high-productivity programming language, where the development time is drastically reduced compared to a compiled language with explicit message-passing.

High-productivity languages such as Matlab and Python – popular languages for scientific computing – are generally accepted as being slower than compiles languages, but more importantly they are inherently sequential and while introducing parallelism is possible in these languages [1][2][3] it limits the productivity. It has been previously shown that it is possible to parallelize matrix and vector-based data-structures from Python, even on distributed memory architectures[4]. However, the parallel execution speed is severely impeded by communication between nodes, in such a scheme for automatic parallelization.

To obtain performance in manual parallelization the programmer usually applies a technique known as latency-hiding, which is a well-known technique to improve the performance and scalability of communication bound problems and is mandatory in many scientific computations.

In this paper, we introduce an abstract model to handle latency-hiding at runtime. The target is scientific applications that make use of vectorized computation. The model enables us to implement latency-hiding into high-productivity programming languages where the runtime system handles communication and parallelization exclusively.

In such high-productivity languages, a key feature is automatic distribution, parallelization and communication that are transparent to the user. Because of this transparency, the runtime system has to handle latency-hiding without any help. Furthermore, the runtime system has no knowledge of the communication pattern used by the user. A generic model for latency-hiding is therefore desirable.

The transparent latency-hiding enables a researcher that uses small self-maintained programs, to use a high-productivity programming language, Python in our case, without sacrificing the possibility of utilizing scalable distributed memory platforms. The purpose of the work is not that the performance of an application, which is written in a high-productivity language, should compete with that of a manually parallelized compiled application. Rather the purpose is to close the gap between high-productivity on a single CPU and high performance on a parallel platform and thus have a high-productivity environment for scalable architectures.

The latency-hiding model proposed in this paper is tailored to parallel programming languages and libraries with the

following properties:

- The programming language requires dynamic scheduling at runtime because it is interpreted.
- The programming language supports and utilizes a distributed memory environment.
- All parallel processes have a global knowledge of the data distribution and computation.
- The programming language makes use of data parallelism in a Single Instruction, Multiple Data (SIMD) fashion in the sense that data affinity dictates the distribution of the computation.

Distributed Numerical Python (DistNumPy) is an example of such a parallel library, and the first project that fully incorporate our latency-hiding model. The implementation of DistNumPy is open-source and freely available[1].

The rest of the paper is organized as follows. In section 2, 3 and 4, we go through the background and theory of our latency-hiding model. In section 5, we describe how we use our latency-hiding model in DistNumPy. In section 6, we present a performance study. Section 7 is future work, and finally in section 8 we conclude.

II. Related Work

Libraries and programming languages that support parallelism in a high productive manner is a well-known concept. In a perfect framework, all parallelism introduced by the framework is completely transparent to the user while the performance and scalability achieved is optimal. However, most frameworks require the user to specify some kind of parallelism – either explicitly by using parallel directives or implicitly by using parallel data structures.

In this paper we will focus on data parallel frameworks, in which parallelism is based on the exploitation of data locality. A classical example of such a framework is High Performance Fortran (HPF) [5], which is an extension of the Fortran-90 programming language. HPF introduces parallelism primarily with vector operations, which, in order to archive good performance, must be aligned by the user to reduce communication. However, a lot of work has been put into eliminating this alignment issue either at compile-time or run-time [6] [7] [8].

DistNumPy[4] is a library for doing numerical computation in Python that targets scalable distributed memory architectures. DistNumPy accomplishes this by extending the NumPy module[9], which combined with Python forms a popular framework for scientific computations. The parallelization introduced in DistNumPy focuses on parallel vector operations like HPF, but because of the latency-hiding we introduce in this paper, it is not a requirement to align vectors in order to achieve good performance.

Hardware architectures also exploit data parallelism to hiding memory latency [10] or communication latency [11]. Likewise, parallel data dependency analysis is essential in order to efficiently schedule instructions and avoid pipeline interlocks [12] [13].

[1]DistNumPy is available at http://code.google.com/p/DistNumPy

III. Latency-Hiding

We define latency-hiding informally as in [14] – "a technique to increase processor utilization by transferring data via the network while continuing with the computation at the same time". When implementing latency-hiding the overall performance depends on two issues: the ability of the communication layer to handle the communication asynchronously and the amount of computation that can overlap the communication – in this work we will focus on the latter issue.

In order to maximize the amount of communication hidden behind computation when performing vectorized computations our abstract latency-hiding model uses a greedy algorithm. The algorithm divides the arrays, and thereby the computation, into a number of fixed-sized data blocks. Since most numerical applications will work on identical dimensioned datasets, the distribution of the datasets will be identical. For many data blocks, the location will therefore be the same and these will be ready for execution without any data transfer. While the co-located data blocks are processed, the transfers of the data blocks from different location can be carried out in the background thus implementing latency-hiding. The performance of this algorithm relies on two properties:

- The number of data blocks must be significantly greater than number of parallel processors.
- A significant number of data blocks must share location.

In order to obtain both properties we need a data structure that support easy retrieval of dependencies between data blocks. Furthermore, the number of data blocks in a computation is proportional with the total problem size thus efficiency is of utmost importance.

IV. Directed Acyclic Graph

It is well-known that a directed acyclic graph (DAG) can be used to express dependencies in parallel applications[15]. Nodes in the DAG represent operations and edges represent serialization dependencies between the operations, which in our case is due to conflicting data block accesses.

Scheduling operations in a DAG is a well-studied problem. The scheduling problem is NP-complete in its general forms [16] where operations are scheduled such that the overall computation time is minimized. There exist many heuristic for solving the scheduling problem [17], but none match our target.

The scheduling problem we solve in this paper is not NP-hard because we are targeting programming frameworks that make use of data parallelism in a SIMD fashion. The parallel model we introduce is statically orchestrating data distribution and parallelization based on predefined data affinity. Assignment of computation tasks are not part of our scheduling problem. Instead, our scheduling problem consists of maximizing the amount of communication that overlaps computation when moving data to the process that is predefined to perform the computation.

In [18] the authors demonstrate that it is possible to dynamic schedule operations in a distributed environment using local DAGs. That is, each process runs a private runtime system and communicates with other processes regarding data dependences. Similarly, our scheduling problem is also dynamic but in our case all processes have a global knowledge of the data distribution and computation. Hence, no communication regarding data dependences is required at all.

The time complexity of insetting a node into a DAG, $G = (V, E)$, is $O(V)$ in worse case. Building the complete DAG is therefore $O(V^2)$. Removing one node from the DAG is $O(V)$, which means that in the case where we simply wants to schedule all operations in a legal order the time complexity is $O(V^2)$. This is without minimizing the overall computation or the amount of communication hidden behind computation. We therefore conclude that a complete DAG approach is inadequate for runtime control of latency-hiding in our case.

We address the shortcoming of the DAG approach through a heuristic that manage dependencies on individual blocks. Instead of having a complete DAG, we maintain a list of depending operations for each data block. Still, the time complexity of scheduling all operations is $O(V^2)$ in worse case, but the heuristic exploits the observation that in the common case a scientific application spreads a vectorized operation evenly between the data blocks in the involved arrays. Thus the number of dependencies associated with a single data block is manageable by a simple linked list. In Section V-G, we will present a practical implementation of the idea.

V. Distributed Numerical Python

The programming language Python combined with the numerical library NumPy[9] has become a popular numerical framework amongst researchers. It offers a high level programming language to implement new algorithms that support a broad range of high level operations directly on vectors and matrices.

The idea in NumPy is to provide a numerical extension to the Python language that enables the Python language to be both high productive and high performing. NumPy provides not only an API to standardized numerical solvers, but also the option to develop new numerical solvers that are both implemented and efficiently executed in Python, much like the idea behind the Matlab[19] framework.

DistNumPy is a new version of NumPy that parallelizes array operations in a manner completely transparent to the user – from the perspective of the user, the difference between NumPy and DistNumPy is minimal. DistNumPy can use multiple processors through the communication library Message Passing Interface (MPI)[20]. However, DistNumPy does not use the traditional single-program multiple-data (SPMD) parallel programming model that requires the user to differentiate between the MPI-processes. Instead, the MPI communication in DistNumPy is fully invisible and the user needs no knowledge of MPI or any parallel programming model. The only difference in the API of NumPy and DistNumPy is the array creation routines. DistNumPy allows both distributed and non-distributed arrays to co-exist, the user must specify, as an optional parameter, if the array should be distributed. The following illustrates the only difference between the creation of a standard array and a distributed array:

```
#Non-Distributed
A = numpy.array([1,2,3])
#Distributed
B = numpy.array([1,2,3], dist=True)
```

A. Views

NumPy and DistNumPy use identical arrays syntax, which is based on the Python list syntax. The arrays are indexed positionally, 0 through length – 1, where negative indexes is used for indexing in the reversed order. Like the list syntax in Python, it is possible to index multiple elements. All indexing that represents more than one element returns a view of the elements rather than a new copy of the elements. This means that an array does not necessarily represent a complete, contiguous block of memory. It is possible to have a hierarchy of arrays where only one array represents a complete contiguous block of memory and the other arrays represent a subpart of that memory. DistNumPy implements an array hierarchy where distributed arrays are represented by the following two data structures.

- **Array-base** is the base of an array and has direct access to the content of the array in main memory. An array-base is created with all related meta-data when the user allocates a new distributed array, but the user will never access the array directly through the array-base. The array-base always describes the whole array and its meta-data such as array size and data type.
- **Array-view** is a view of an array-base. The view can represent the whole array-base or only a sub-part of the array-base. An array-view can even represent a non-contiguous sub-part of the array-base. An array-view contains its own meta-data that describes which part of the array-base is visible. The array-view is manipulated directly by the user and from the users perspective the array-view is simply a normal contiguous array.

Array-views are not allowed to refer to each other, which means that the hierarchy is flat with only two levels: array-base below array-view. However, multiple array-views are allowed to refer to the same array-base. This two-tier hierarchy is illustrated in Figure 1.

B. Data Layout

The data layout in DistNumPy consists of three kinds of data blocks: base-blocks, view-blocks and sub-view-blocks, which make up a three level abstraction hierarchy (Fig. 2).

- **Base-block** is a block of an array-base. It maps directly into one block of memory located on one node. The memory block is not necessarily contiguous but only one MPI-process has exclusive access to the block. Furthermore, DistNumPy makes use of a N-Dimensional Block Cyclic

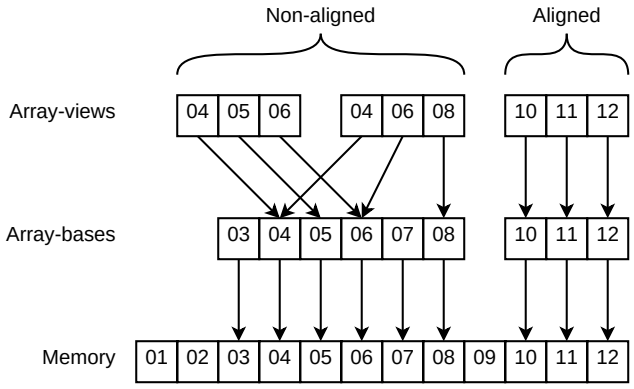


Fig. 1. Reference hierarchy between the two array data structures and the main memory. Only the three array-views at top of the hierarchy are visible from the perspective of the user.

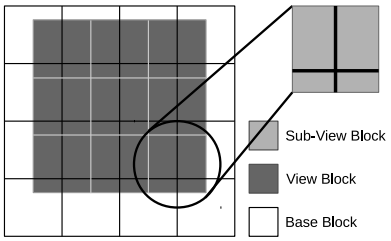


Fig. 2. An illustration of the block hierarchy that represents a 2D distributed array. The array is divided into three block-types: Base, View and Sub-View-blocks. The 16 base-blocks make up the base-array, which may be distributed between multiple MPI-processes. The 9 view-blocks make up a view of the base-array and represent the elements that are visible to the user. Each view-block is furthermore divided into four sub-view-blocks, each located on a single MPI-process.

Distribution inspired by High Performance Fortran[5], in which base-blocks are distributed across multiple MPI-processes in a round-robin fashion.

- **View-block** is a block of an array-view, from the perspective of the user a view-block is a contiguous block of array elements. A view-block can span over multiple base-blocks and consequently also over multiple MPI-processes. For a MPI-process to access a whole view-block it will have to fetch data from possibly remote MPI-processes and put the pieces together before accessing the block. To avoid this process, which may cause some internal memory copying, we divide view-blocks into sub-view-block.
- **Sub-view-block** is a block of data that is a part of a view-block but is located on only one MPI-process. The driving idea is that all array operation is translated into a number of sub-view-block operations.

We will define an *aligned array* as an array that have a direct, contiguous mapping through the block hierarchy. That is, a distributed array in which the base-blocks, view-blocks and sub-view-blocks are identical. A *non-aligned array* is then a distributed array without this property.

C. Universal Function

An important mechanism in DistNumPy is a concept called a Universal function. A universal function (ufunc) is an element-wise vector operation that computes all elements in an array-view independently. Applying an ufunc operation on a whole array-view is semantically equivalent to performing the ufunc operation on each array-view block individually. This property makes it possible to perform a distributed ufunc operation in parallel. A distributed ufunc operation consists of four steps:

1) All MPI-processes determine the distribution of the view-block computation, which is strictly based on the distribution of the output array-view.
2) All MPI-processes exchange array elements in such a manner that each MPI-process can perform its computation locally.
3) All MPI-processes perform their local computation.
4) All MPI-processes send the altered array elements back to the original locations.

D. Latency-Hiding

The standard approach to hide communication latency behind computation in message-passing is a technique known as double buffering. The implementation of double buffering is straightforward when operating on a set of data block that all have identical sizes. The communication of one data block is overlapped with the computation of another already communicated data block and since the sizes of all the data blocks are identical all iterations are identical.

In DistNumPy, a straightforward double buffering approach works well for ufuncs that operate on aligned arrays, because it translates into communication and computation operations on whole view-blocks, which does not benefit from latency-hiding inside view-blocks. However, for ufuncs that operate on non-aligned arrays this is not the case because the view-block is distributed between multiple MPI-processes. In order to achieve good scalable performance the DistNumPy implementation must therefore introduce latency-hiding inside view-blocks. For example the computation of a view-block in Figure 2 can make use of latency-hiding by first initiating the communication of the three non-local sub-view-blocks then compute the local sub-view-block and finally compute the three communicated sub-view-blocks.

E. Operation Dependencies

One of the key contributions in this paper is a latency-hiding model that, by maintaining data dependencies between scheduled operations, is able to aggressively initiate communication and lazily evaluate tasks, in order to allow maximal time for the communication to finish before entering a wait state. In this section, we will demonstrate the idea of the model by giving an example of a small 3-point stencil computation implemented in DistNumPy (Fig. 3). For now, we will use a traditional DAG for handling the data dependencies. Later we will describe the implementation of the heuristic that enables us to manage dependencies more efficiently. Additionally, it should be noted

```
1 M = numpy.array([1,2,3,4,5,6],dist=True)
2 N = numpy.empty((6),dist=True)
3 A = M[2:]
4 B = M[0:4]
5 C = N[1:5]
6 C = A + B
```

Fig. 3. This is an example of a small 3-point stencil application.

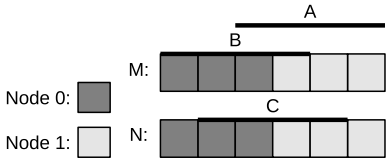


Fig. 4. The data layout of the two arrays M and N and the three array-views A, B and C in the 3-point stencil application (Fig. 3). The arrays are distributed between two nodes using a block-size of three.

that the parallel processes do not need to exchange dependency information since they all have full knowledge of the data distribution.

Two processes are executing the stencil application and DistNumPy distributes the two arrays, M and N, using a block-size of three. This means that three contiguous array elements are located on each process (Fig. 4). Using a DAG as defined in section IV, figure 5 illustrates the dependencies between 12 operations that together constitute the execution. Initially the following six operations are ready:

$$R := \{op1, op2, op3, op4, op9, op10\}$$

Afterwards, without the need of communication, two more operations $op5$ and $op8$ may be executed. Thus, it is possible to introduce latency-hiding by initiating the communication, $op6$ and $op7$, before evaluating operation $op5$ and $op8$. The amount of communication latency hidden depends on the computation time of $op5$ and $op8$ and the communication time of $op6$ and $op7$.

We will strictly prioritize between operations based on whether they involve communication or computation – giving priority to communication over computation. Furthermore, we

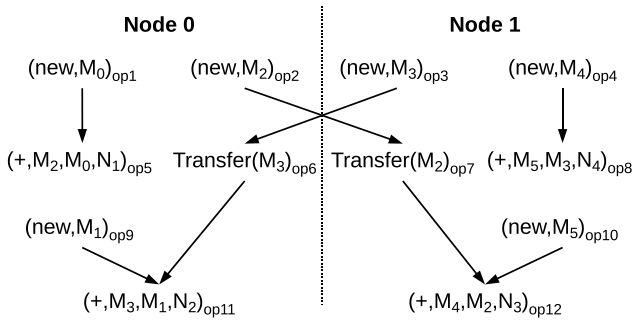


Fig. 5. Illustration of a DAG that represents the dependencies in a 3-point stencil application (Fig. 3). The DAG consists of 12 operations, *op1 to op12*, divided between two processes.

will assume that all operations take the same amount of time, which is a reasonable assumption in DistNumPy since it divides array operations into small blocks that often have the same computation or communication time.

F. Lazy Evaluation

Since Python is an interpreted dynamic programming language, it is not possible to schedule communication and computation operations at compile time. Instead, we introduce lazy evaluation as a technique to determine the communication and computation operations used in the program at runtime.

During the execution of a DistNumPy program all MPI-processes record the requested array operations rather than applying them immediately. The MPI-processes maintain the operations in a convenient data structure and at a later point all MPI-processes apply the operations. The idea is that by having a set of operations to carry out it may be possible to schedule communication and computation operations that have no mutual dependencies in parallel.

We will only introduce lazy evaluation for Python operations that involve distributed arrays. If the Python interpreter encounters operations that do not include DistNumPy arrays, the interpreter will execute them immediately. At some point, the Python interpreter will trigger DistNumPy to execute all previously recorded operation. This mechanism of executing all recorded operation we will call an *operation flush* and the following three conditions may trigger it.

- The Python interpreter issues a read from distributed data. E.g. when the interpreter reaches a branch statement.
- The number of delayed operations reaches a user-defined threshold.
- The Python interpreter reaches the end of the program.

G. The Dependency System

The main challenge when introducing lazy evaluation is to implement a dependency system that schedules operations in a performance efficient manner while the implementation keeps the overhead at an acceptable level.

Our first lazy evaluation approach makes use of a DAG-based data structure to contain all recorded operations. When an operation is recorded, it is split across the sub-view-blocks that are involved in the operation. For each such operation, a DAG node is created just as in figure 3 and 4.

Beside the DAG our dependency system also consist of a *ready queue*, which is a queue of recorded operations that do not have any dependencies. The ready queue makes it possible to find operations that are ready to be executed in the time complexity of $O(1)$.

a) Operation Insertion: The recording of an operation triggers an insertion of new node into the DAG. A straightforward approach will simply implement insertion by comparing the new node with all the nodes already located in the DAG. If a dependency is detected the implementation adds an edge between the nodes. The time complexity of such an implementation is $O(n)$ where n is the number of operation in the DAG and the construction of the complete DAG is $O(n^2)$.

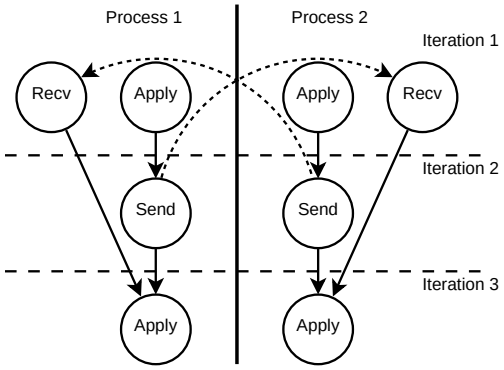


Fig. 6. Illustration of the naïve evaluation approach. The result is a deadlock in the first iteration since both processes are waiting for the receive-node to finish, but that will never happen because the matching send-node is in second iteration.

b) Operation Flush: To achieve good performance the operation flush implementation must maximize the amount of communication that it is overlapped by computation. Therefore, the flush implementation initiate communication at the earliest point in time and only do computation when all communication has been initiated. Furthermore, to make sure that there is progress in the MPI layer it checks for finished communication in between multiple computation operations. The following is the flow of our operation flush algorithm:

1) Initiate all communication operations in the ready queue.
2) Check in a non-blocking manner if some communication operations have finished and remove finished communication operations from the ready queue and the DAG. Furthermore, register operations that now have no dependencies into the ready queue.
3) If there is only computation operations in the ready queue execute one of them and remove it from the ready queue and the DAG.
4) Go back to step one if there are any operations left in the ready queue else we are finished.

The algorithm maintains the following three invariants:

1) All operations, that are ready, are located in the ready queue.
2) We only start the execution of a computation node when there is no communication node in the ready queue.
3) We only wait for communication when the ready queue has no computation nodes.

1) Deadlocks: To avoid deadlocks a MPI-process will only enter a blocking state when it has initiated all communication and finished all ready computation. This guaranties a deadlock-free execution but it also reduces the flexibility of the execution order. Still, it is possible to check for finished communication using non-blocking functions such as `MPI_Testsome()`.

The naïve approach to evaluate a DAG is simply to first evaluate all nodes that have no dependencies and then remove the evaluated nodes from the graph and start over – similar to the traditional BSP model. However, this approach may result in a deadlock as illustrated in figure 6.

2) Dependency Heuristic: Experiments with lazy evaluation using the DAG-based data structure shows that the overhead associated with the creation of the DAG is very time consuming and becomes the dominating performance factor. We therefore introduce a heuristic to speed up the common case. We base the heuristic on the following two observations:

- In the common case, a scientific DistNumPy application spreads a computation evenly between all sub-view-blocks in the involved arrays.
- Operation dependencies are only possible between sub-view-blocks that are part of the same base-block.

The heuristic is that instead of having a DAG, we introduce a prioritized operation list for each base-block. The assumption is that, in the common case, the number of operations associated with a base-block is manageable by a linked list.

We implement the heuristic using the following algorithm. A number of operation-nodes and access-nodes represent the operation dependencies. The operation-node contains all information needed to execute the operation on a set of sub-view-blocks and there is a pointer to an access-node for each sub-view-block. The access-node represents memory access to a sub-view-block, which can be either reading or writing. E.g., the representation of an addition operation on three sub-view-blocks is two read access-nodes and one write access-node (Fig. 7).

Our algorithm places all access-nodes in dependency-lists based on the base-block that they are accessing. When an operation-node is recorded each associated access-node is inserted into the dependency list of the sub-view-blocks they access. Additionally, the number of accumulated dependencies the access-nodes encounter is saved as the operation-node's reference counter.

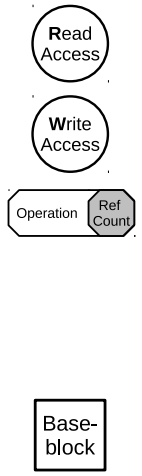
All operation-nodes that are ready for execution have a reference count of zero and are in the ready queue. Still, they may have references to access-nodes in dependency-lists – only when we execute an operation-node will we remove the associated access-nodes from the dependency-lists. Following the removal of an access-node we traverse the dependency-list and for each depending access-node we reduce the associating reference counter by one. Because of this, the reference counter of another operation-node may be reduced to zero, in which case we move the operation-node to the ready queue and the algorithm starts all over.

Figure 7 goes through all the structures that make up the dependency system and figure 8 illustrates a snapshot of the dependency system when executing the 3-point stencil application.

VI. Experiments

To evaluate the performance impact of the latency-hiding model introduced in this paper, we will conduct performance benchmark using DistNumPy and NumPy[2]. The benchmark is executed on an Intel Core 2 Quad cluster (Table I) and for each

[2]NumPy version 1.3.0



All access-nodes that access the same base-block are linked together in a dependency-list. The order of the list is simply based on the time of node insertion (descending order). Additionally an access-node contains the information needed to determine whether it depends on other access-nodes.

An operation-node has a pointer to all access-nodes that are involved in the operation. Attached to an operation is a reference counter that specifies the number of dependencies associated with the operation. When the counter reaches zero the operation is ready for execution. At some point when the operation has been executed the operation-node and all access-nodes are completely remove from the dependency system.

A base-block simply contains a pointer to the first access-node in the dependency-list.

Fig. 7. The structures used in the dependency system.

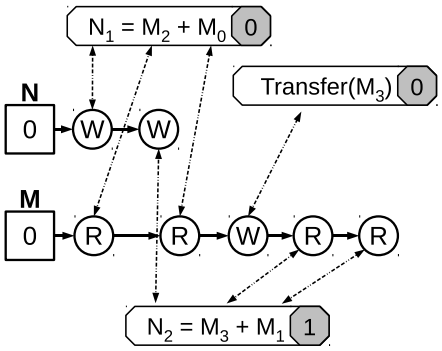


Fig. 8. Illustration of the dependency system when executing the 3-point stencil in figure 3, 4 and 5. The illustration is a snapshot of the dependency system on node 0 after the creation of all the arrays. Note that since the block size is three, node 0 only has one block of each array.

application we calculate the speedup of DistNumPy compared to NumPy. The problem size is constant though all the executions, i.e. we are measuring strong scaling. To measure the performance impact of the latency-hiding, we use two different setups: one with latency-hiding and one that uses blocking communication. For both setups we measured the time spent on waiting for communication, i.e. the communication latency not hidden behind computation.

In this benchmark we utilizes the cluster in a *by node* fashion. That is, from one to sixteen CPU-cores we start on MPI-process per node (no SMP) and above sixteen CPU-cores we start multiple MPI-processes per node. The MPI library

TABLE I
HARDWARE SPECIFICATIONS

CPU	Intel Xeon E5345
CPU Frequency	2.33 GHz
CPU per node	2
Cores per CPU	4
Memory per node	16 GB
Number of nodes	16
Network	Gigabit Ethernet

```
# Black Scholes Function
#  S: Stock price
#  X: Strike price
#  T: Years to maturity
#  r: Risk-free rate
#  v: Volatility
def BlackScholes(CallPutFlag,S,X,T,r,v):
 d1 = (log(S/X)+(r+v*v/2.)*T)/(v*sqrt(T))
 d2 = d1-v*sqrt(T)
 if CallPutFlag=='c':
   return S*CND(d1)-X*exp(-r*T)*CND(d2)
 else:
   return X*exp(-r*T)*CND(-d2)-S*CND(-d1)
```

Fig. 9. This is the Black Sholes Function in the **Black-Scholes** benchmark where `CND` is the cumulative normal distribution. Note that there is no source code difference between a parallel and a sequential version – it is regular Python/Numpy source code.

used throughout this benchmark is OpenMPI[3].

The benchmark consists of the following six Python applications.

- **Fractal** Computation of the Mandelbrot Set. From a NumPy tutorial written by Walt[21] (Fig. 11).
- **Black-Scholes** Computation of the Black-Scholes model[22] implemented in NumPy (Fig. 9 and 12).
  Both **Fractal** and **Black-Scholes** are embarrassingly parallel applications and we expect that latency-hiding will not improve the performance.
- **N-body** A Newtonian N-body simulation that uses a naïve algorithm that computes all body-body interactions. The NumPy implementation is a translation of a MATLAB application by Casanova[23] (Fig. 13).
- **kNN** A naïve implementation of a k nearest neighbor search (Fig. 14).
  The **N-body** and **kNN** applications have a computation complexity of $O(n^2)$. This indicates that the two applications should have good scalability even without latency-hiding.
- **Jacobi** The Jacobi method is an algorithm for determining the solutions of a system of linear equations. It is an iterative method that uses a spitting scheme to approximate the result (Fig. 15).
- **Jacobi Stencil** In this benchmark, we have implemented the Jacobi method using stencil operations rather than matrix row operations (Fig. 10 and 16).
  The two **Jacobi** applications also have a computation complexity of $O(n)$. However, the constant associated with n is very small, e.g. to compute one element in the Jacob Stencil application four adjacent elements are required. We expect latency-hiding to be very important for good scalability.

A. Discussion

Overall, the benchmarks show that DistNumPy has both good performance and scalability. However, the scalability is somewhat worsening at 32 CPU-cores and above, which

[3]OpenMPI version 1.5.1

```
1 while epsilon < delta:
2   T = 0.2 * (A[1:-1,1:-1] + A[2:,1:-1] \
3             + A[1:-1,0:-2] + A[1:-1,2:])
4   delta = sum(abs(A[1:-1,1:-1] - T))
5   A[1:-1,1:-1] = T
```

Fig. 10. This is the kernel in the **Jacobi Stencil** benchmark. Note that there is no source code difference between a parallel and a sequential version – it is regular Python/Numpy source code.

correlates with the use of multiple CPU-cores per node. Because of this distinct performance profile, we separate the following discussion into results executed on one to sixteen CPU-cores (one CPU-core per node) and the results executed on 32 CPU-cores to 128 CPU-cores (multiple CPU-cores per node).

1) One to Sixteen CPU-cores: The benchmarks clearly shows that DistNumPy has both good performance and scalability. Actually, half of the Python applications achieve super-linear speedup at sixteen CPU-cores. This is possible because DistNumPy, opposed to NumPy, will try to reuse memory allocations by lazily de-allocating arrays. DistNumPy uses a very naïve algorithm that simply checks if a new array allocation is identical to a just de-allocated array. If that is the case one memory allocation and de-allocation is avoided.

In the two embarrassingly parallel applications, **Fractal** and **Black-Scholes**, we see very good speedup as expected. Since the use of communication in both applications is almost non-existing the latency-hiding makes no difference. The speedup achieved at sixteen CPU-cores is 18.8 and 15.4, respectively.

The two naïve implementations of **N-body** and **kNN** do not benefit from latency-hiding. In **N-body** the dominating operations are matrix-multiplications, which is a native operation in NymPy and in DistNumPy implemented as specialized operations using the parallel algorithm SUMMA[24] and not as a combination of ufunc operations. Since both the latency-hiding and the blocking execution use the same SUMMA algorithm the performance is very similar. However, because of the overhead associated with latency-hiding, the blocking execution performs a bit better. The speedup achieved at sixteen CPU-cores is 17.2 with latency-hiding and 17.8 with blocking execution. Similarly, the performance difference between latency-hiding and blocking in **kNN** is minimal – the speedup achieved at sixteen CPU-cores is 12.5 and 12.6, respectively. The relatively modest speedup in **kNN** is the result of poor load balancing. At eight and sixteen CPU-cores the chosen problem is not divided evenly between the processes.

Finally, latency-hiding introduces a drastically improved performance to the two communication intensive applications **Jacobi** and **Jacobi Stencil**. The waiting time percentage going from 54% to 2% and from 62% to 9%, respectively. Latency-hiding also has a major impact on the overall speedup, which goes from 5.9 to 12.8 and from 7.7 to 18.4, respectively. In other words latency-hiding more than doubles the overall speedup and CPU utilization.

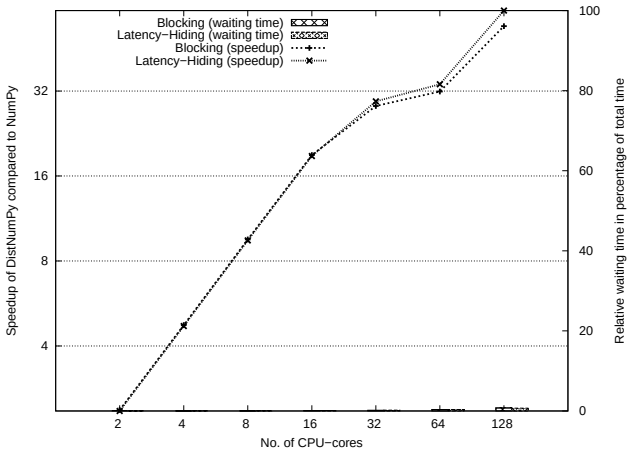


Fig. 11. Speedup of the Fractal application.

2) Scaling above sixteen CPU-cores: Overall, the performance is worsening at 32 CPU-cores – particular at 64 CPU-cores and above where the CPU utilization is below 50%. One reason for this performance degradation is the characteristic of strong scaling. In order to have considerable more data blocks than MPI-processes, the size of the data distribution blocks decreases as the number of executing CPU-cores increases. Smaller data blocks reduces the performance since the overhead in DistNumPy is proportional with the size of a data block.

However, smaller data blocks are not enough to justify the observed performance degradation. The von Neumann bottleneck[25] associated with main memory also hinder scalability. This is evident when looking at Figure 17, which is a speedup graph of **N-body** that compares *by node* and *by core* scaling. At eight CPU-cores, both result uses identical data distribution and block size, but the performance when only using one CPU-core per node is clearly superior to using all eight CPU-cores on one node.

A NumPy application will often use ufuncs heavily, which makes the application vulnerable to the von Neumann bottleneck. The problem is that multiple ufunc operations are not pipelined in order to utilize cache locality. Instead, NumPy will compute a single ufunc operation at a time. This problem is also evident in DistNumPy since our latency-hiding model only address communication latency and not memory latency.

VII. Future Work

The latency-hiding model introduced in this paper focuses on communication latency. However, the result from out benchmarks shows that memory latency is another aspect that is important for good scalability – particular when utilizing shared memory nodes.

One approach to address this issue is to extend our latency-hiding model with cache locality optimization. The scheduler will have to prioritize computation operations that are likely to be in the cache. One heuristic to accomplish this is to sort the operations in the ready queue after the last time the associated data block has been accessed.

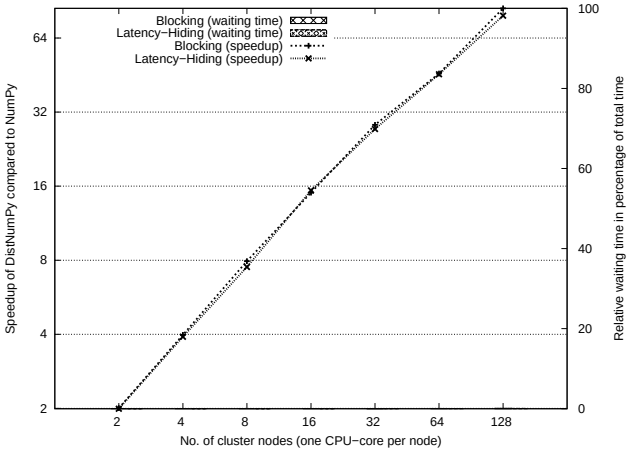


Fig. 12. Speedup of the Black-Scholes application.

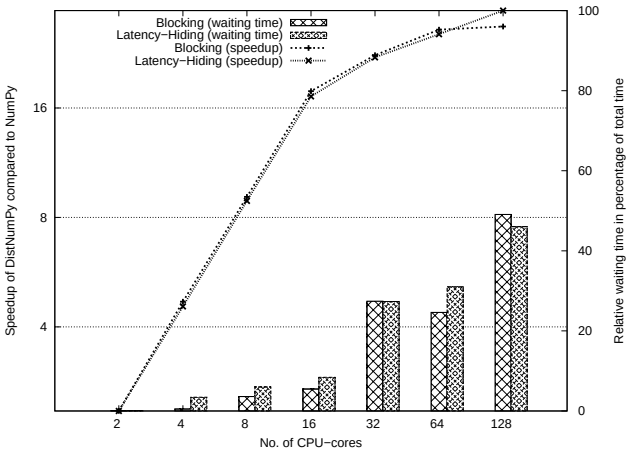


Fig. 13. Speedup of the N-body application.

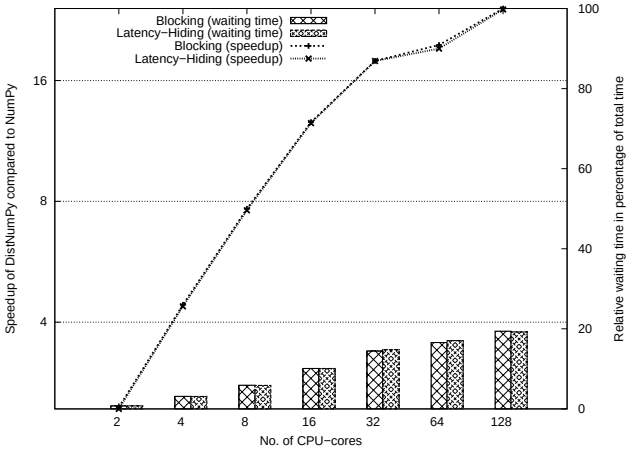


Fig. 14. Speedup of the kNN application.

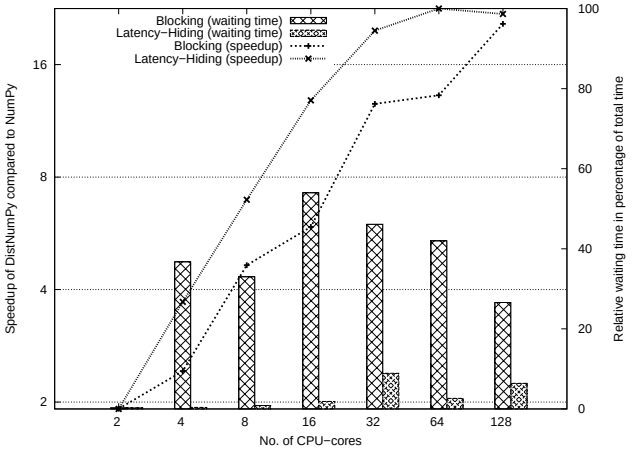


Fig. 15. Speedup of the Jacobi application.

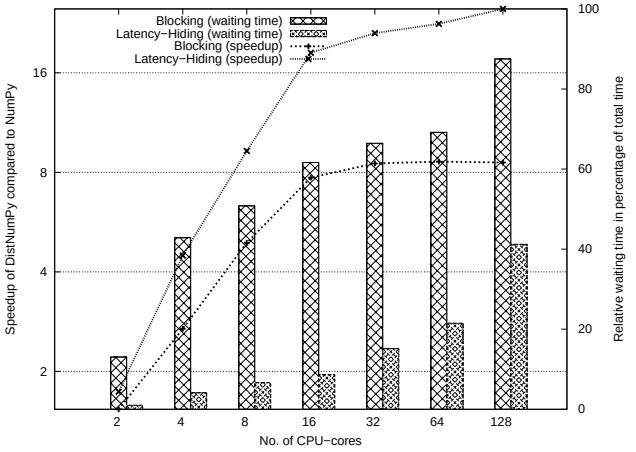


Fig. 16. Speedup of the Jacobi Stencil application.

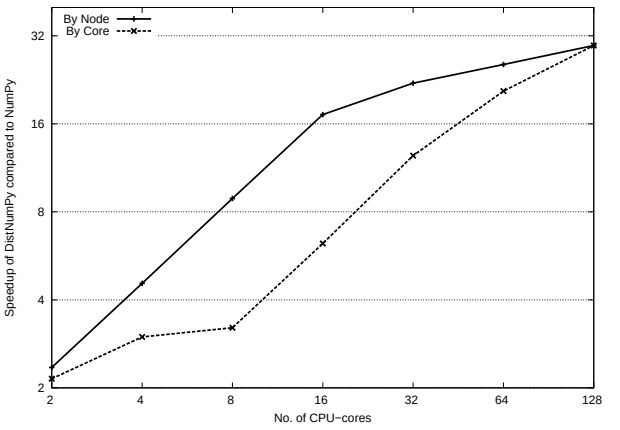


Fig. 17. Speedup of the N-body application that compares *by node*, in which the maximum number of CPU-cores is used, and *by core*, in which the minimum number of nodes is used. Note that the *by node* graph is identical to the *latency-hiding* graph in figure 13.

Another approach is to merge calls to ufuncs, that operate on common arrays, together in one joint operation and thereby make the joint operation more CPU-intensive. If it is possible to merge enough ufuncs together the application may become CPU bound rather than memory bound.

Introducing Hybrid Programming could also be a solution to the problem. In order to utilize hybrid architectures, [26] shows that shared and distributed memory programming can improve the overall performance and scalability.

VIII. Conclusions

While automatic parallelization for distributed memory architectures cannot hope to compete with a manually parallelized version, the productivity that comes with automatic parallelization still makes the technique of interest to a user who only runs a code a few times between changes. For applications that are embarrassingly parallel or applications where the computational complexity is $O(n^2)$ or higher, it is fairly straight forward to manage the communication for automatic parallelization. However, for common kernels the complexity is $O(n \log(n))$ or even $O(n)$ and here the application of latency-hiding techniques is essential for performance.

In this work we have presented a scheme for managing latency-hiding, that is based on the assumption that splitting the work in more blocks than there are processors will allow us to aggressively communicate data-blocks between nodes, while at the same time processing operations that require no external data-blocks. The same dependency analysis may be done without a division into data-blocks, but the blocking approach allows us to maintain a full DAG, an operation that is known to be costly, and replace the DAG with a number of ordered linked lists, to which access is done in linear time.

We implement the model in Distributed Numerical Python, DistNumPy, a programming framework that allows linear algebra operations expressed in NumPy to be executed on distributed memory platforms and this is without any effort towards parallelization from the programmer.

A selection of six benchmarks show that the system, as predicted, does not improve the performance of embarrassingly parallel applications or applications with complexity $O(n^2)$ or higher. For applications with lower complexity the benefit from automatic latency-hiding is highly dependent on the relationship between the amount of data that needs to be transferred and the cost of updating the individual elements. The performance of the stencil-based Jacobi-solver improves from a speedup of 7.7 to 18.4 at sixteen processors and 8.6 to 25.0 at 128 processors, compared to standard sequential NumPy. This is matched by the fact that the time spend on waiting for communication drops from 62% to 9% and 87% to 41%, respectively, with the introduction of latency-hiding.

Overall, the conclusion is that managing latency-hiding at runtime is fully feasible and makes automatic parallelization feasible for a number of applications where manual parallelization would otherwise be required. The most obvious target is the large base of stencil-based algorithms.

References

[1] J. Hollingsworth, K. Liu, and V. P. Pauca, "Parallel toolbox for matlab," Winston-Salem, NC, USA, Tech. Rep., 1996.

[2] O. J. Anshus, J. M. Bjørndalen, and B. Vinter, "Pycsp - communicating sequential processes for python," in *Communicating Process Architectures 2007*, A. A. McEwan, W. Ifill, and P. H. Welch, Eds., jul 2007, pp. 229–248.

[3] R. Bromer and F. Hantho, "pupympi," 2011. [Online]. Available: https://bitbucket.org/bromer/pupympi

[4] M. R. B. Kristensen and B. Vinter, "Numerical python for scalable architectures," in *Fourth Conference on Partitioned Global Address Space Programming Model, PGAS'10*. ACM, 2010.

[5] D. Loveman, "High performance fortran," *IEEE Parallel & Distributed Technology: Systems & Applications*, vol. 1, no. 1, p. 25, 1993.

[6] C. Koelbel and P. Mehrotra, "Compiling global name-space parallel loops for distributed execution," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, no. 4, pp. 440–451, 1991.

[7] T. Brandes and F. Zimmermann, "Adaptor - a transformation tool for hpf programs," 1994.

[8] S. Benkner, H. Zima, P. Mehrotra, and J. V. Rosendale, "High-level management of communication schedules in hpf-like languages," Tech. Rep., 1997.

[9] T. E. Oliphant, "Python for scientific computing," *Computing in Science and Engineering*, vol. 9, pp. 10–20, 2007.

[10] R. Espasa, M. Valero, and J. E. Smith, "Out-of-order vector architectures," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 30. Washington, DC, USA: IEEE Computer Society, 1997, pp. 160–170.

[11] S. Kaxiras, "Distributed vector architectures," *Journal of Systems Architecture*, vol. 46, no. 11, pp. 973–990, 2000.

[12] J. L. Hennessy and T. Gross, "Postpass code optimization of pipeline constraints," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 3, pp. 422–448, 1983.

[13] P. B. Gibbons and S. S. Muchnick, "Efficient instruction scheduling for a pipelined architecture," *SIGPLAN Not.*, vol. 21, no. 7, pp. 11–16, 1986.

[14] V. Strumpen and T. L. Casavant, "Exploiting communication latency hiding for parallel network computing: Model and analysis," in *Proc. PDS'94*. IEEE, 1994, pp. 622–627.

[15] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques and tools*. Reading, MA: Addison-Wesley, 1986.

[16] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[17] A. A. Khan, C. L. McCreary, and M. S. Jones, "A Comparison of Multiprocessor Scheduling Heuristics," *Parallel Processing, International Conference on*, vol. 2, pp. 243–250, 1994.

[18] F. Song, A. YarKhan, and J. Dongarra, "Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM.

[19] M. U. Guide, "The mathworks," *Inc., Natick, MA*, vol. 5, 1998.

[20] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI Portable Parallel Programming with the Message Passing Interface*. The MIT Press, 1994.

[21] S. van der Walt, "Numpy: lock 'n load," 2008. [Online]. Available: http://mentat.za.net/numpy/intro/intro.html

[22] F. Black and M. Scholes, "The pricing of options and corporate liabilities," *The journal of political economy*, pp. 637–654, 1973.

[23] H. Casanova, "N-body simulation assignment," Nov 2008. [Online]. Available: http://navet.ics.hawaii.edu/~casanova/courses/ics632_fall08/projects.html

[24] R. A. v. d. Geijn and J. Watts, "Summa: scalable universal matrix multiplication algorithm," *Concurrency - Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[25] J. Backus, "Can programming be liberated from the von neumann style?: A functional style and its algebra of programs," *Communications of the ACM*, vol. 16, no. 8, pp. 613–641, 1978.

[26] M. Kristensen, H. Happe, and B. Vinter, "Hybrid parallel programming for blue gene/p," *Scalable Computing: Practice and Experience*, vol. 12, no. 2, 2011.

A.6 PGAS for Distributed Numerical Python Targeting Multi-core Clusters

Mads Ruben Burgdorff Kristensen, Yili Zheng, and Brian Vinter. PGAS for Distributed Numerical Python Targeting Multi-core Clusters
In Proceedings of the 2012 IEEE International Symposium on Parallel and Distributed Processing (IPDPS'12). IEEE.

PGAS for Distributed Numerical Python Targeting Multi-core Clusters

Mads Ruben Burgdorff Kristensen
Niels Bohr Institute
University of Copenhagen
Denmark
madsbk@nbi.dk

Yili Zheng
Lawrence Berkeley National Lab
Berkeley, CA 94720
USA
yzheng@lbl.gov

Brian Vinter
Niels Bohr Institute
University of Copenhagen
Denmark
vinter@nbi.dk

***Abstract*—In this paper we propose a parallel programming model that combines two well-known execution models: Single Instruction, Multiple Data (SIMD) and Single Program, Multiple Data (SPMD). The combined model supports SIMD-style data parallelism in global address space and supports SPMD-style task parallelism in local address space. One of the most important features in the combined model is that data communication is expressed by global data assignments instead of message passing. We implement this combined programming model into Python, making parallel programming with Python both highly productive and performing on distributed memory multi-core systems.**

We base the SIMD data parallelism on DistNumPy, an auto-parallelizing version of the Numerical Python (NumPy) package that allows sequential NumPy programs to run on distributed memory architectures.

We implement the SPMD task parallelism as an extension to DistNumPy that enables each process to have direct access to the local part of a shared array. To harvest the multi-core benefits in modern processors we exploit multi-threading in both SIMD and SPMD execution models. The multi-threading is completely transparent to the user – it is implemented in the runtime with OpenMP and by using multi-threaded libraries when available.

We evaluate the implementation of the combined programming model with several scientific computing benchmarks using two representative multi-core distributed memory systems – an Intel Nehalem cluster with Infiniband interconnects and a Cray XE-6 supercomputer – up to 1536 cores. The benchmarking results demonstrate scalable good performance.

***Keywords*-Parallel Programming; Parallel Computing; Python; Scientific Computing;**

I. Introduction

Finding the solution for computational scientific and engineering problems often requires experimenting various algorithms and different parameters with feedback in several iterations. Therefore, being able to quickly prototype the solution is critical to timely and successful scientific discovery. Python has emerged as such an important programming language in the Computational Science and Engineering community [1], [2], [3] for its ease of use, concise syntax close to mathematical formulas and the rich set of packages available. While Python is popular and widely used on individual workstations, it has limited support for developing parallel algorithms on large scale distributed-memory clusters. Currently, the most common usage of Python for supercomputers is to provide a convenience wrapper to run legacy parallel applications written in FORTRAN and C with MPI. However, it is challenging to implement parallel applications directly through Python due to the constraint of separate physical address spaces on distributed-memory systems. In addition, there are concerns about slow execution of the Python interpreter for HPC. To make quickly prototyping parallel algorithms on supercomputers possible, we extend Python with global view support on distributed-memory systems.

A. SIMD and SPMD

There exists a broad range of execution models for parallel programming. Single Instruction, Multiple Data (SIMD) and Single Program, Multiple Data (SPMD) are two execution models often used in parallel programming. Our SIMD execution model refers to a single sequential Python instruction stream with massive data parallelism. Our programming model is executed in SIMD mode from the user's perspective but the underlying runtime system is built on top on MPI and is executed in SPMD mode.

SIMD is well suited for expressing data-centric parallelism and avoids many nasty parallel programming bugs, such as deadlocks and data races, due to the single thread execution model. In our distributed-memory implementation, the SIMD programming model provides full knowledge of data distribution and dependencies to all participating processors, which makes it is possible for the runtime system to execute array operations and perform data movement in parallel without user interventions. Additionally, the processors need not communicate when performing data dependency analysis and operation scheduling optimizations at runtime. However, SIMD also reduces programming flexibility because the user is restricted to data-parallel operations. The SPMD execution model provides more flexibility than SIMD by allowing different processes to execute different code paths. But it makes auto parallelization and auto communication much harder because each process only has it local state information.

B. DistNumPy

Distributed Numerical Python (DistNumPy) [4] is an extended version of NumPy [1] that parallelizes array operations for distributed-memory systems in a completely transparent manner from the user's perspective. DistNumPy can use multiple processors through the Message Passing Interface (MPI) [5] communication library. The original DistNumPy only uses the SIMD execution model with implicit data parallelism in which the MPI communication is fully invisible to the user. The only difference in the API of NumPy and DistNumPy is the array creation routines. DistNumPy allows both distributed and non-distributed arrays to co-exist and the user specifies, as an optional parameter, if the array should be distributed (Lst. 1).

Listing 1: Creation of Local and Global Arrays

```
1 #Non-Distributed
2 A = numpy.array([1,2,3])
3 #Distributed
4 B = numpy.array([1,2,3], dist=True)
```

Figure 1 shows an example of using global matrix views to implement a stencil operation by just one line of DistNumPy code. In contrast, conventional programming languages would require using tedious scalar operations with for loops and MPI send/recv to express the same high-level operation.

C. SPMD Extension to DistNumPy

In this paper, we introduce an extension to DistNumPy that mixes the already existing SIMD execution model in DistNumPy with the SPMD execution model. This new extension enables users to express parallel algorithms in terms of global data management and local operations. Using LINPACK as an example, the user may express the distributed-memory LU factorization algorithm in Python and use BLAS/LAPACK for local computations.

To overcome the relatively slow execution of the Python interpreter, we use four optimization techniques to amortize the overheads:

- Multi-threading with OpenMP to exploit data parallelism in array operations.
- Use optimized libraries for basic local computations whenever possible, such as BLAS, LAPACK, FFTW and vendor-optimized libraries. In common cases, most execution time would be spent on computation in the library and thus the overheads incurred by the Python interpreter are negligible. Even for applications written in FORTRAN and C, it is a good practice to use optimized libraries if available because they usually run much faster than the standard implementations.
- Combine array operations through lazy evaluation. Using an internal dependency tracking system, the DistNumPy runtime system will aggregate operations and execute them in batch only when the results are required in the data flow. This lazy evaluation strategy can not only perform code optimization on-the-fly but also minimize the overheads of executing individual Python instructions.
- Overlap computation and communication by leveraging non-blocking communication. With sufficient overlapping, the Python interpreter overhead is hidden and will not increase the execution time.

In case all fails, the user still has the option to implement the performance-critical section in low-level languages and use it in the Python code to speed up execution. Because the Python implementation is more concise to understand, it is much easier to identify the bottleneck in such program than searching it in a very large C or FORTRAN application.

The authors in [6] optimize DistNumPy by combining array operations through lazy evaluation and overlapping computation and communication. We use this optimized version of DistNumPy as the bases of our work. For a detailed description of the lazy evaluation and automatic communication latency hiding, we refer to prior work [4], [6].

We have incorporated our extension into the open-source DistNumPy package[1] and demonstrated the feasibility of using Python to implement parallel algorithms for multi-core clusters. Specifically, the main contributions of this paper include:

- Added support for SIMD and SPMD execution models in a single parallel program so that both data-parallel and task-parallel applications can be conveniently implemented with Python.
- Improved parallelism and scalability of DistNumPy by implementing hierarchical parallelism in the runtime: using OpenMP for multi-threading and MPI for inter-node communication.
- Enabled interoperability between DistNumPy and existing third-party libraries for efficient local computation.
- Developed three benchmarks with our proposed SIMD+SPMD programming model and evaluated their performance on up to 1536 cores.

II. Related Work

Our programming system supports both the SIMD data-parallel programming model and the SPMD task-parallel programming model. High Performance Fortran (HPF) [7] and ZPL [8] are two well-known examples of data-parallel programming languages. HPF is a Fortran-based data-parallel programming language that requires static compilation for distributed-memory systems [9]. Our Python-based programming system uses dynamic execution and on-the-fly data dependency analysis, which is more accurate than static analysis and thus enable more performance optimizations by reordering Python interpreter instructions at runtime.

[1]DistNumPy is available at http://code.google.com/p/DistNumPy

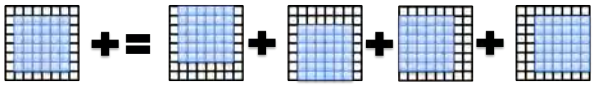


Figure 1: Matrix expression of a simple 5-point stencil computation example. See listing 2, line 8 for the Python expression.

Global Arrays (GA) [10], [11] is a library that supports global arrays in SPMD programs for distributed-memory computers. The main difference between Global Arrays and our programming system is that our DisNumPy supports the SIMD execution model, of which the single control flow is more convenient for data parallelism and easier to reason about for programmers. In addition, our programming system provides nice object encapsulations for the array objects and hence allows the user to naturally perform array operations, for example, writing $A + B$ for matrix addition.

Python extensions, NumPy [1] and SciPy [2], have been successfully used in scientific computing [3], [12] because their high-level abstractions are very close to mathematical formulas and there exist a super rich set of Python packages for almost any common task. Both NumPy and SciPy are targeted to single-node systems but our extension is targeted to multi-node systems.

MPI is the most popular parallel programming model for distributed memory systems today and there are several implementations for providing MPI to Python programs, such as MPI for Python [13]. In contrast to message passing, our programming system enables the user to directly access shared data through assignment statements and our runtime system performs communication automatically based on the location of the data.

Many recent parallel programming languages, such as UPC, Co-array Fortran, X10 and Chapel [14], [15], [16], [17], [18], [19], [20], provide the feature of Partitioned Global Address Space (PGAS), which combines the efficiency of leveraging data locality and the programming productivity of using shared-memory. Our programming system based on Python belongs to the PGAS family and supports global shared data across all processes and private local data within each process.

III. Programming Model

We propose a new PGAS programming model that combines the SIMD execution model for global array operations and the SPMD execution model for local array operations. In this section, we first briefly introduce the basic array syntax used in NumPy and DistNumPy and then describe our programming model.

A. NumPy and DistNumPy Syntax

NumPy and DistNumPy use identical array syntax based on the Python *l*ist syntax. Elements of 1-D arrays are indexed from 0 to $(length - 1)$, where negative integers are used to index in the reversed order. Multi-dimensional arrays are stored in row-major order and indexed by each dimension sequentially. Like the list syntax in Python, it is possible to index multiple elements with ranges. Array ranges including more than one elements return a view (reference) to the sub-array rather than a new copy of the elements. Operations performed on the view change the underlying data of the original array directly. This view mechanism makes it convenient to implement the stencil operation in Figure 1. NumPy and DistNumPy have a `copy` method to create a real copy of the array data when needed.

Listing 2: 5-Point-Stecil

```
1  center = full[1:-1, 1:-1]
2  up     = full[0:-2, 1:-1]
3  down   = full[2:  , 1:-1]
4  left   = full[1:-1, 0:-2]
5  right  = full[1:-1, 2:  ]
6  while epsilon < delta:
7    work[:] = center
8    work += 0.2 * (up+down+left+right)
9    diff = absolute(center - work)
10   delta = sum(diff)
11   center[:] = work
```

Listing 3: Cannon's algorithm

```
1  for i in xrange(nrow / BS):
2    #Apply local matrix multiplication
3    C.local()[:] += np.dot(A.local(), B.local())
4
5    #Moving columns horizontal (left)
6    tmp = A[:,0:BS].copy()
7    A[:,0:-BS] = A[:,BS:]
8    A[:,-BS:] = tmp
9
10   #Moving rows vertical (up)
11   tmp = B[0:BS,:].copy()
12   B[0:-BS,:] = B[BS:,:]
13   B[-BS:,:] = tmp
```

B. Global Array Operations

DistNumPy supports global arrays distributed among all available processes. Python applications can make use of DistNumPy by creating such global arrays. All global array operations use the SIMD execution model, in which all processes need to execute the same Python statement sequence even if some of them don't need to take actions. The computation place is based on the "owner computes" rule. Processes that own part or all the operands of an operation need to participate in the operation. Non-participating processes would simply mask out the current Python statement.

In DistNumPy, each process runs a Python interpreter that interprets the Python application. However, because of the synchronous nature of SIMD, the parallelism provided

by DistNumPy is completely transparent to the user when every interpreter takes the same code paths and only uses global arrays operation. This transparency enables a user familiar with Python/NumPy to utilize multiple processes seamlessly. Listing 2 shows the kernel of a 5-point-stencil DistNumPy application where all arrays are global and all participating Python interpreters will execute identical code. The parallel code is identical to the sequential NumPy version and the data-parallel operations are automatically executed in parallel on distributed-memory computers.

C. Global Assignment Operations

The SPMD execution model typically uses explicit message passing (one or two sided) to move data in a distributed environment. DistNumPy enables using global data assignments to express data movement. For example, when multiplying two matrices by Cannon's algorithm, we shift matrix-blocks in the vertical and horizontal directions. Listing 3 line 5-13, is an implementation of this data movement using regular Python assignments. Actual communication associated with the assignments is completely hidden to the user. The user only needs to specify the algorithm's data dependencies and DistNumPy handles how to perform the data movements.

D. Local Array Operations

The user can get the local part of a global array from DistNumPy by the `local` function, which is the only global array operation that is non-collective and does not imply synchronization. The `local` function returns a NumPy array view that can be used together with any NumPy compatible Python libraries.

Listing 3 is an implementation of Cannon's algorithm where we assume the matrices are square. The implementation makes use of the local address space to compute the local matrix multiplication (line 3) and the global address space to move matrix blocks up and left (line 5-13). The user can mix local and global operations as needed. The only rule is that a local array becomes undefined when executing a global array operation. The user needs to use `local` to retrieve a new local NumPy array after a global array operation . This restriction is because of the lazy evaluation used in the original DistNumPy (Sec. IV).

E. Local Array Block Iterator

Iterating over all local array blocks is a common operation. `blocks` is an operation that returns such an iterator. It is semantically equivalent to `local` but instead of returning the local part of the global array, it returns an iterator that iterates over all local blocks in the global array.

F. Data Layout

The user may have to be aware of the data layout when using local array operations. In the SPMD execution model, the user will often have to differentiate computations based on the process identity and data layout. To facilitate this, we assign a rank id to each process that is accessible through the constant `RANK`. The user may specify the data layout for global arrays at job start-up time and it is immutable over the course of the execution.

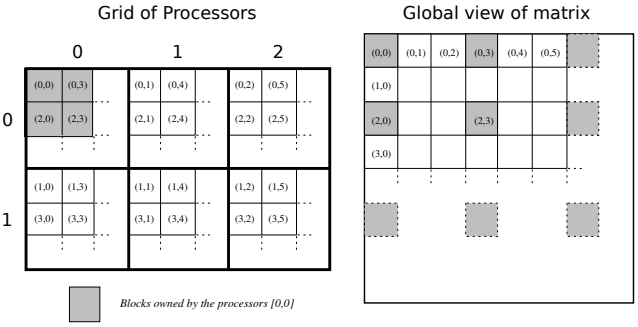


Figure 2: The Two-Dimensional Block Cyclic Distribution of a matrix on a 2 x 3 grid of processors.

DistNumPy global arrays are stored in generalized N-Dimensional block cyclic distribution schemes inspired by HPF [7]. The user can define flexible process grids by using the `datalayout()` function. The block size is the same for all global arrays and is accessible through the constant `BLOCKSIZE`. Fig. 2) is an example of the common 2-D block cyclic data layout for matrix. The distribution scheme works by arranging all processes in a two dimensional grid and then distributing data-blocks in a round-robin fashion either along one or both grid dimensions.

G. One Process Distribution

DistNumPy supports an alternative array distribution where all data is located on a single process. When creating a distributed array it is possible to specify where the data is located by a rank affinity parameter. The following operation will create a global array that is located on process 42 exclusively:

```
A = np.array([1,2,3], dist=True, onenode=42)
```

IV. Implementation

DistNumPy is extended from NumPy as a Python package that can be used with regular Python interpreters. DistNumPy uses dynamic data dependency analysis, lazy evaluation and communication aggregation techniques to hide communication latency [6]. Following the data dependencies between batched operations, DistNumPy proactively initiates data transfers as early possible while consumes the data as late as possible to maximize overlapping between communication and computation.

A. Lazy Evaluation

Since Python is an interpreted dynamic programming language, it is not possible to schedule communication and computation operations at compile time. Instead, DistNumPy determines the best execution order of operations in the program with lazy evaluation at run time. During the execution of a DistNumPy program, all processes record the requested array operations in a convenient data structure and then apply them in batch later with optmizations.

DistNumPy only uses lazy evaluation for Python operations that involve global arrays. If the Python interpreter encounters operations that do not include global arrays, the interpreter will execute them immediately. The Python interpreter will trigger DistNumPy to execute all previously recorded operation at any of the following conditions:

- The Python interpreter executes a `local` operation.
- The number of delayed operations passes a user-defined threshold.
- The Python interpreter reaches the end of the program.

In order to achieve good performance when executing all previously recorded operations, DistNumPy tries to maximize communication and computation overlapping. Therefore, DistNumPy initiates non-blocking communication at the earliest time and only does computation after all communication has been initiated. Furthermore, DistNumPy checks for communication completion between multiple computation operations to make sure that there is progress in the communication layer. The execution flow is as follows:

1) Initiate all non-depended communication operations.
2) Check if any communication operations has been finished in a non-blocking manner and insert operations that have no dependencies into the ready queue.
3) When only computation operations are ready, execute one of them and move new operations that have no dependencies into the ready queue.
4) Go back to step one if there are unfinished operations or else terminate.

The algorithm maintains the following three invariants:

1) All ready operations are in the ready queue.
2) Computation operations are executed only when there is no communication operation in the ready queue.
3) Communication operations are checked for completion when there is no computation operation in the ready queue.

B. Multi-threading with OpenMP

The lack of efficient multi-threading support in the original DistNumPy is a severe limitation when executing on multi-core distributed memory systems. We improve the performance for data-parallel array operations by using multi-threading with OpenMP.

In NumPy and DistNumPy, a universal function (ufunc) is a vectorized function that operates on all array elements independently and provides implicit data-parallelism. In the runtime, we use OpenMP directives to parallelize the *for* loops in the ufunc computations.

The current multi-threading implementation harvests parallelism within single operation instead of parallelism over multiple operations because we find that it requires extensive dependency analyses to do so, which is beyond the scope of this paper.

C. Third Party Python Libraries

There exist a great number of optimized numerical Python libraries. Most of them are compatible with NumPy because NumPy provides a C-pointer to the raw array data. DistNumPy can also make use of these NumPy libraries for local computations by converting the local part of the global array to a regular NumPy array. For example, in the Cannons algorithm we use the local NumPy function `dot()` (Lst. 3, line 3), which is a simple binding to an optimized BLAS library, to compute local matrix multiplication.

V. Benchmarks

To evaluate the implementation of our proposed programming model, we developed three mini-benchmarks: 1) matrix multiplication, 2) LU factorization and 3) 2-D heat equation solution with the Jacobi iterative method. We implemented all three benchmarks in pure Python and then used third party libraries, BLAS and LAPACK, to compute local results when applicable.

A. Matrix Multiplication

Matrix multiplication is a fundamental operation in numerical computations. The global matrices are distributed across all nodes with 2-D block-cyclic date layout. We use the SUMMA [21] algorithm, which is based on outer-product BLAS level-3 updates implemented by row and column broadcast communication followed by local matrix multiplications on each node.

Listing 4 shows the complete source code of the SUMMA implementation. It is completely written in Python and uses the NumPy function `np.dot()` (line 26) to compute the local matrix multiplication. `np.dot()` calls the optimized BLAS library available on the system.

The only communication in the SUMMA algorithm is in line 15-19, in which we replicate a column-block horizontally and a row-block vertically. The communication is elegantly expressed by global array assignments to `a_work` and `b_work`.

Listing 4: SUMMA Matrix Multiplication

```
1 import numpy as np
2
3 def summa(a,b,c):
4   (prow,pcol) = a.pgrid()
5   BS = np.BLOCKSIZE
6   a_work = np.zeros((a.shape[0],BS*pcol), \
7                     dtype=a.dtype, dist=True)
```

```
8   b_work = np.zeros((BS*prow,b.shape[1]), \
9                     dtype=a.dtype, dist=True)
10  Ksz = a.shape[1]
11  for k in xrange(0,Ksz,BS):
12    bs = min(BS, Ksz-k)#Current block size
13
14    #Replicate column-block horizontal
15    for p in xrange(pcol):
16      a_work[:,p*BS:p*BS+bs] = a[:,k:k+bs]
17    #Replicate row-block vertical
18    for p in xrange(prow):
19      b_work[p*BS:p*BS+bs,:] = b[k:k+bs,:]
20
21    #Apply local outer dot product
22    l_a_work = a_work.local()[:,:bs]
23    l_b_work = b_work.local()[:bs,:]
24    l_c = c_new.local()
25    if l_c.size > 0:
26      l_c += np.dot(l_a_work, l_b_work)
```

B. LU Factorization

LU factorization is a classical numerical linear algebra problem that decomposes a matrix as the product of a lower triangular matrix and an upper triangular matrix. The implementation is a straightforward translation of a block LU factorization written in Matlab [22]. Our implementation is simplified by skipping partial pivoting. Listing 5 shows the complete source code of implementation, which makes use of the SUMMA implementation (line 47) from before (Lst. 4) and a local LU factorization (line 25) provide by the SciPy library, which in turn uses an optimized LAPACK library. Taking advantage of the distributed array extension, we express all communication patterns through Python assignments when replicating the local LU results horizontally and vertically (line 27-31).

Listing 5: LU Factorization

```
1  import numpy as np
2  from scipy import linalg
3  import pyHPC
4  from itertools import izip as zip
5
6  def lu(matrix):
7    SIZE = matrix.shape[0]
8    BS = np.BLOCKSIZE
9
10   (prow,pcol) = matrix.pgrid()
11   A = matrix.copy()
12   L = np.zeros((SIZE,SIZE),dtype=matrix.dtype,\
13             dist=True)
14   U = np.zeros((SIZE,SIZE),dtype=matrix.dtype,\
15                 dist=True)
16   for k in xrange(0,SIZE,BS):
17     bs = min(BS,SIZE - k) #Current block size
18     kb = k / BS # k as block index
19
20     #Compute local LU
21     slice = ((kb,kb+1),(kb,kb+1))
22     for a,l,u in zip(A.blocks(slice), \
23                      L.blocks(slice), \
24                      U.blocks(slice)):
25       (p,l[:],u[:]) = linalg.lu(a)
26
27     #Replicate local LU horizontal and vertical
28     for tk in xrange(k+bs,SIZE,BS):
29       tbs = min(BS,SIZE - tk)
30       L[tk:tk+tbs,k:k+bs] = U[k:k+tbs,k:k+bs]
31       U[k:k+bs,tk:tk+tbs] = L[k:k+bs,k:k+tbs]
32
33     if k+bs < SIZE:
34       #Compute horizontal multiplier
35       slice = ((kb,kb+1),(kb+1,SIZE/BS))
36       for a,u in zip(A.blocks(slice), \
37                      U.blocks(slice)):
38         u[:] = np.linalg.solve(u.T,a.T).T
39
40       #Compute vertical multiplier
41       slice = ((kb+1,SIZE/BS),(kb,kb+1))
42       for a,l in zip(A.blocks(slice), \
43                      L.blocks(slice)):
44         l[:] = np.linalg.solve(l,a)
45
46       #Apply to remaining submatrix
47       A -= pyHPC.summa(L[:,:k+bs],U[:k+bs,:])
48
49   return (L, U)
```

C. Heat Equation

The heat equation benchmark is to solve a partial differential equation that describes the distribution of heat in a given region over time. We use the Jacobi iterative method to approximate the result with a 5-point stencil implementation. Listing 2 shows the computation loop of the implementation, which is concisely expressed by array operations and assignments.

VI. Performance

We evaluate the performance of our benchmarks on two representative multi-core distributed memory systems – an Intel Nehalem cluster with Infiniband interconnects and a Cray XE-6 supercomputer – up to 1536 cores (Table I).

Both systems consist of multi-core Non-Uniform Memory Access (NUMA) shared-memory nodes and each node has multiple NUMA domains. CPU cores within the same NUMA domain have uniform data access latency to the local memory while CPU cores of different NUMA domains would have non-uniform data access latencies. We use hybrid parallelism, processes with threads, for all of our benchmark runs. Specifically, we run one process per NUMA domain and one thread per core within the NUMA domain. Threads within a NUMA domain communicate through shared memory and processes across NUMA domains communicate through MPI.

Through empirical study, we find that this is the configuration that achieved best performance. The two extremes usually do not work well: 1) running one process per core without threading causes too much overheads in terms of both memory footprint and communication time; 2) running one process per node and using threads across NUMA domains would also slow down the execution due to the NUMA issues with data locality and resource contention with too many threads.

System	Intel Infiniband Cluster	Cray XE-6
Processor	Intel Xeon X5530	AMD Opteron 6172
Clock	2.4 GHz	2.1 GHz
Peak Performance per Core	10.6 Gflops	8.4 Gflops
Cores per NUMA Domain	4	6
NUMA Domains per Node	2	4 (packaged in 2 sockets)
Total Cores per Node	8	24
Private L1 Data Cache	64 KB	64 KB
Private L2 Data Cache	512 KB	512 KB
Shared L3 Cache per Socket	8MB	12MB
Memory Bandwidth	25.6 GB/s	25.6 GB/s
Memory per Node	24GB DDR3-1066 ECC	32GB DDR3-1066 ECC
Compiler	Intel C/C++ 11.1	PGI 11.3
Math Library	Intel MKL 10.2	Cray Scientific Library 10.5
Interconnect	Infiniband 4X QDR	Gemini 3-D Torus
Peak Bandwidth (per direction)	5 GB/s	7 GB/s
MPI	OpenMPI 1.4.2	Cray MPI 5.1.4

Table I: Two distributed-memory multi-core NUMA systems for the experiments

For each benchmark, we calculate the FLOPS based on the floating operation counts of the ideal sequential algorithm and the measured execution times. Additionally, we compare the results with the linearly scaling performance, which we calculate by extrapolating the sequential FLOPS performance of NumPy. We use this comparison as an upper bound of the achievable scalable performance. We perform weak scaling experiments, in which the problem size is scaled with the number of CPU-cores in the executions.

A. Matrix Multiplication (SUMMA)

Matrix multiplication has very high computation to communication ratio: its asymptotic computation complexity is $O(n^3)$ while its data communication complexity is only $O(n^2)$, as verified in the communication and computation ratio Figure 4. The SUMMA algorithm is a well-known scalable parallel algorithm for matrix multiplication. Thus this benchmark scales nearly linearly on the Intel Infiniband cluster and also performs quite well on the Cray XE-6 system as in Figure 3. The communication cost of running the benchmark on CPU cores in a single NUMA domain is zero because we use threads to share data directly as described before.

Because all local matrix computations are done with the vendor-optimized BLAS libraries (Intel MKL and Cray Scientific Library), our implementation obtains very good absolute performance in terms of the hardware peak FLOP rate on both platforms. The performance of our Python sequential implementation is very close to that of the C sequential implementation because in both cases the majority of the running time is spent in external optimized BLAS library.

The gap between our implementation "DistNumPy" and the ideal linear speed-ups of the sequential implementation is basically the overheads of performing parallel execution with the high-level abstractions in our programming model. For the Intel Infiniband cluster, the python interpreter overheads are negligible as our obtained performance tightly tracks the linear speed-up curve in Figure 3 (a).

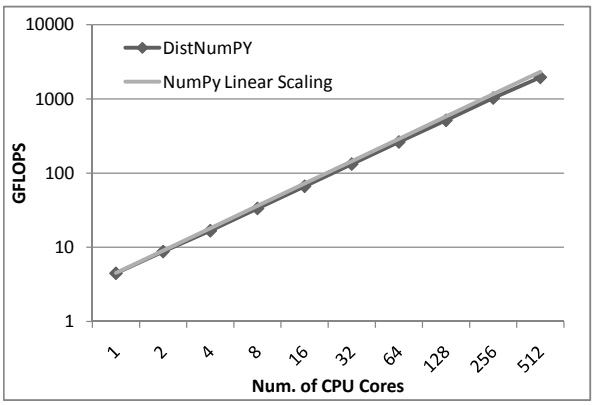
B. LU Factorization

The LU factorization benchmark also has relatively high computation to communication ratio but does require more communication and has more data dependencies than the matrix multiplication benchmark. Thus it is expected to scales well. Figure 5 shows that the LU benchmark scale well on the Intel Infiniband cluster up to 512 cores. But the scalability of the LU benchmark decreases on the Cray XE-6 system when using more than 384 cores when the communication times become dominant, as shown in Figure 6. The Python sequential implementation of LU is about 12% and 50% slower than the C counterpart on the Intel Infiniband cluster and the Cray XE-6 system respectively. The Python implementation is slower because it performs extra work to allocate buffers and format data in addition to calling the LAPACK factorization routine.

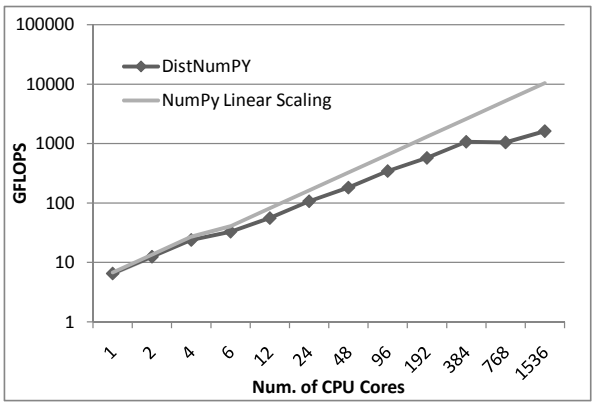
Overall, the benchmark performance is better on the Intel Infiniband cluster than on the Cray XE-6 system as the current Cray MPI for the Cray Gemini network has limited overlapping support for non-blocking MPI communication. In addition, the job scheduler on the Cray system may allocate distant nodes to a job and the torus network performance would suffer from the communication traffics caused by other jobs.

C. Heat Equation

The computation to communication ratio of this stencil-type benchmark is inherently low, which is a small constant. Thus its performance is somewhat limited by the memory bandwidth when running within a node with shared-memory and limited by the network latency and bandwidth when running on multiple nodes. Figure 7 shows that our implementation scales well on up to 256 cores on the Intel Infiniband cluster and up to 192 cores on the Cray XE-6 system. As the number of cores goes up, the performance

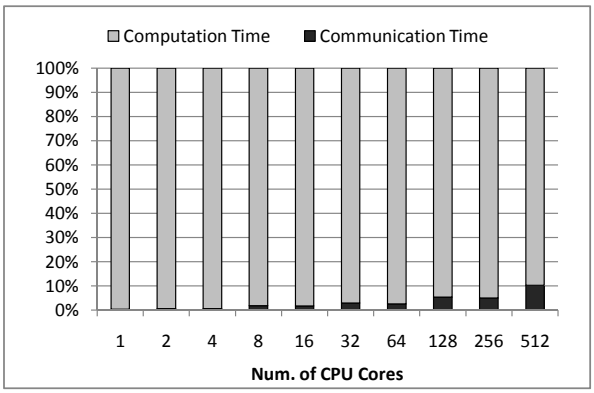


(a) Intel Infiniband cluster

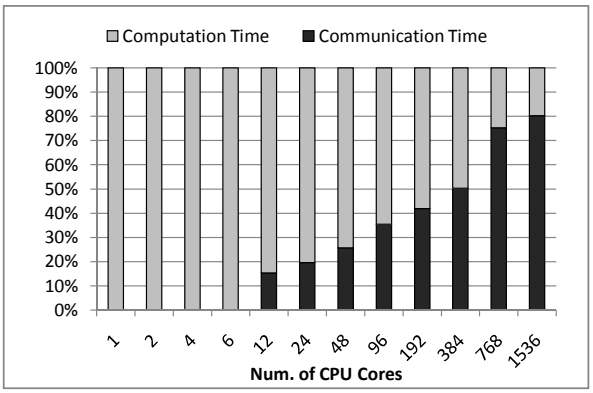


(b) Cray XE-6

Figure 3: Matrix Multiplication (SUMMA) benchmark performance

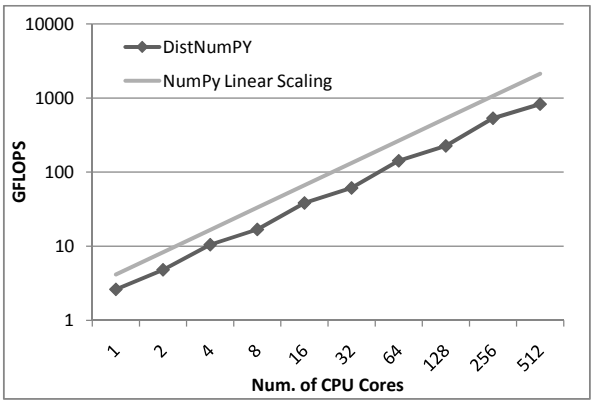


(a) Intel Infiniband cluster

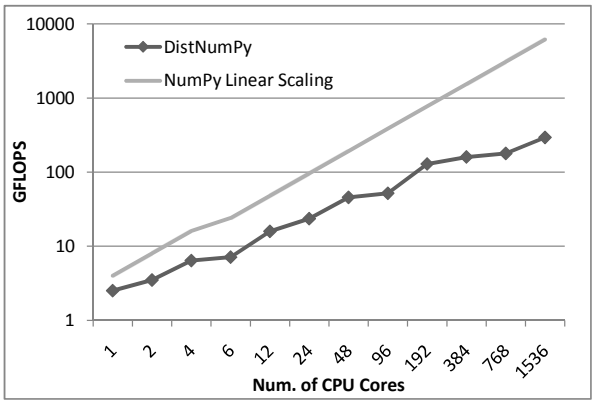


(b) Cray XE-6

Figure 4: Matrix Multiplication (SUMMA) benchmark communication and computation ratio

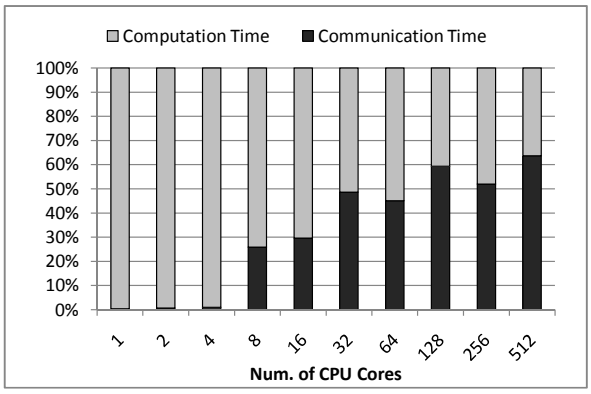


(a) Intel Infiniband cluster

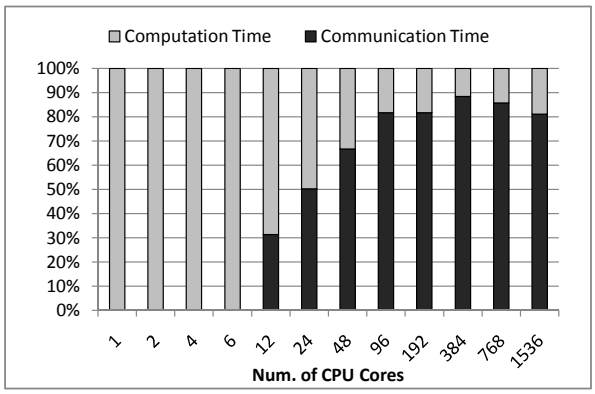


(b) Cray XE-6

Figure 5: LU Factorization benchmark performance

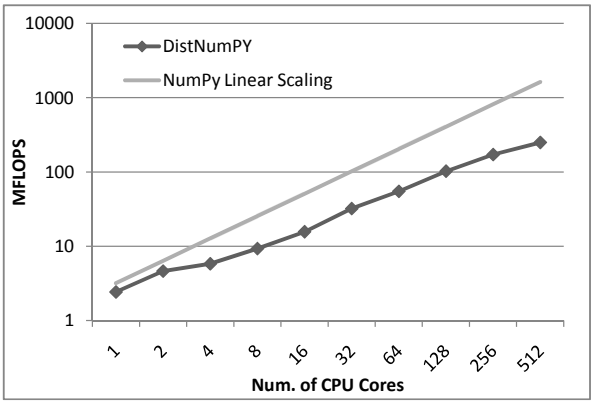


(a) Intel Infiniband cluster

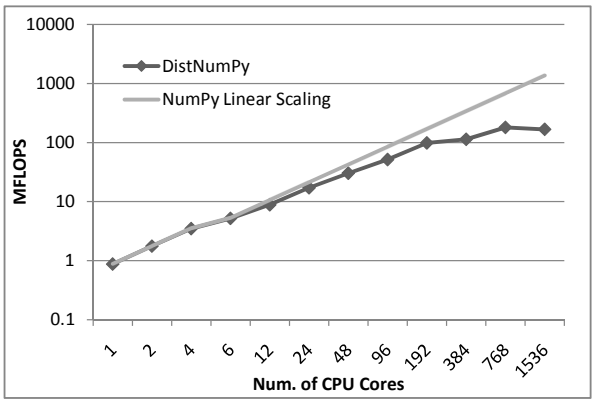


(b) Cray XE-6

Figure 6: LU Factorization communication and computation ratio

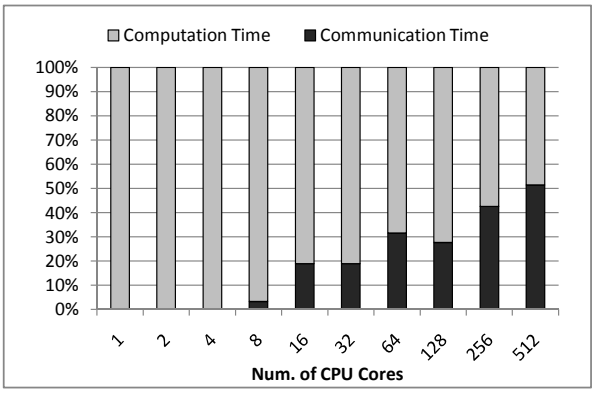


(a) Intel Infiniband cluster

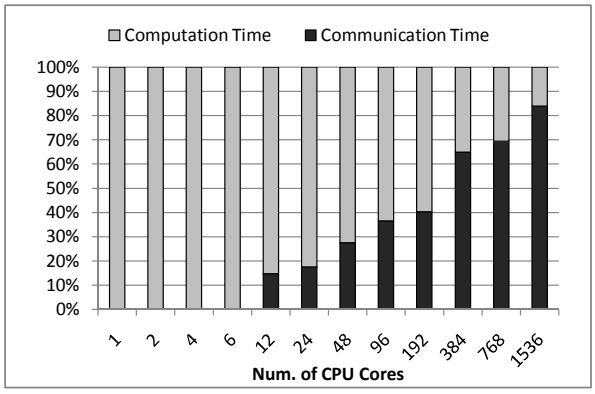


(b) Cray XE-6

Figure 7: Heat Equation benchmark performance



(a) Intel Infiniband cluster



(b) Cray XE-6

Figure 8: Heat Equation benchmark communication and computation ratio

is increasingly dominated by the communication time which results in suboptimal scalability (Fig. 8). The C implementation of the Heat Equation benchmark performs much better than the Python implementation and we plan to address this performance discrepancy issue due to Python interpreter overheads in future work.

D. Scalability Limitations

The global address space and the SIMD execution model introduce some scalability limitations. It is nice to express data movement using the global address space but it forces each process to iterate over all elements in a global array operation not only the elements it has to compute. Thus, introducing a Python overhead that increases proportional with the global size of an array operation. In a traditional SPMD execution model the overhead is proportional with the local size of an array operation.

Another important limitation is the lack of communication latency-hiding. DistNumPy will normally use lazy evaluation to overlap communication with computation. However, the amount of instructions lazy evaluated decreases when applications use the `local` operation.

VII. Future Work

One of the main features in DistNumPy when using the SIMD execution model is automatic communication latency hiding. This feature is disabled somewhat when mixing SIMD and SPMD. The problem is that DistNumPy has to execute all lazy evaluated operations when the user changes to the SPMD model, e.g. when the user calls `local` in order to apply computations based on process affinity. Therefore, DistNumPy will not automatically overlap local operations with communication.

The introduction of a *scheduler* function in DistNumPy could solve the problem. The user would then use this new function to schedule local operations instead of applying them immediately. This would enable DistNumPy to include the local operation in its lazy evaluation system thus making it possible to overlap local operations with communication.

One the other hand, it is harder to address the scalability limitation introduced by using the global address space. The use of global data iterations is very natural when programming using the global address space. Still, it is possible to limit this issue by introducing dedicate global functions, such as functions to move or replicate data.

Both the Matrix Multiplication and LU factorization benchmark replicate data blocks, listing 4 line 14-19 and listing 5 line 27-31, by using Python loops that iterates over global arrays. A new function similar to the matrix replication function, `repmat`, in Matlab could handle these replications with a single function call thus making the Python loops unnecessary.

VIII. Conclusion

The single execution flow with fully synchronous operations of SIMD is both the main strength and weakness of data-parallel programming models: two most notorious types of parallel programming bugs, data races and deadlocks, simply don't exist in data-parallel programs because there is only one execution thread. However, it is inconvenient to perform task parallelism and conditional computations with pure data parallelism. To get the benefits of both data-parallelism and task-parallelism, we incorporate both SIMD and SPMD execution models in our programming system.

Python and other scripting languages are commonly considered as unsuitable for large scale high performance parallel computing due to its interpreter execution nature. Our work is a proof of concept that shows that Python with proper extensions and optimizations can indeed be used to develop parallel applications that scale on large distributed memory systems. The loss of raw performance due to Python interpreter overheads is considerably small because the major part of execution time is spent in the underlying computation and communication libraries. In addition, significant speed-ups can often be achieved by using better algorithms and refined models, which are made much easier to implement due to the high-level abstractions of our Python-based programming system. Like Python, our programming model is general and applicable beyond scientific computing applications. A distributed shared array in our system is essentially a partitioned global address space in that the array elements may be used to store arbitrary objects.

As the ExaScale Software Study [23] pointed out: "current software approaches will be inadequate in enabling future Grand Challenge applications on Extreme Scale systems". Traditional parallel programming languages, such as C and Fortran, are good for getting hardware performance but less suitable for high-level application development and algorithm exploration. We propose a two layer approach to address the programming challenges for extreme scale systems: a low-level language layer that provides portable performance across different hardware platforms and a high-level language layer that provides portable productivity to end users. The work presented in this paper is a step towards creating such a high-level programming system and has demonstrated the feasibility of achieving productivity without compromising performance.

Acknowledgment

This research was supported in part by the Office of Science of the U.S. Department of Energy (DE-AC02-05CH11231). This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] T. E. Oliphant, "Python for scientific computing," *Computing in Science and Engineering*, vol. 9, pp. 10–20, 2007. [Online]. Available: http://numpy.scipy.org

[2] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online]. Available: http://www.scipy.org/

[3] P. F. Dubois, "Guest editor's introduction: Python: Batteries included," *Computing in Science Engineering*, vol. 9, no. 3, pp. 7 –9, may-june 2007.

[4] M. R. B. Kristensen and B. Vinter, "Numerical python for scalable architectures," in *Fourth Conference on Partitioned Global Address Space Programming Model, PGAS'10*. ACM, 2010.

[5] MPI Forum, "MPI: A message-passing interface standard, v1.1," University of Tennessee, Knoxville, Technical Report, June 12, 1995.

[6] M. Ruben Burgdorff Kristensen and B. Vinter, "Managing Communication Latency-Hiding at Runtime for Parallel Programming Languages and Libraries," *Arxiv Preprint arXiv:1201.3804v1* , jan 2012.

[7] High Performance Fortran Forum, "High Performance Fortran language specification, version 1.0," Houston, Tex., Tech. Rep. CRPC-TR92225, 1993. [Online]. Available: citeseer.ist.psu.edu/fortran92high.html

[8] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and D. Weathersby, "ZPL: A machine independent programming language for parallel computers," *Software Engineering*, vol. 26, no. 3, pp. 197–211, 2000. [Online]. Available: citeseer.ist.psu.edu/article/chamberlain00zpl.html

[9] K. Kennedy, C. Koelbel, and H. Zima, "The rise and fall of high performance fortran: an historical object lesson," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, ser. HOPL III. New York, NY, USA: ACM, 2007, pp. 7–1–7–22. [Online]. Available: http://doi.acm.org/10.1145/1238844.1238851

[10] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apr, "Advances, applications and performance of the global arrays shared memory programming toolkit," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 203–231, Summer 2006. [Online]. Available: http://hpc.sagepub.com/content/20/2/203.abstract

[11] "Global arrays webpage," http://www.emsl.pnl.gov/docs/global/.

[12] T. Oliphant, "Python for scientific computing," *Computing in Science Engineering*, vol. 9, no. 3, pp. 10 –20, may-june 2007.

[13] "MPI for Python," http://mpi4py.scipy.org/.

[14] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen, "Productivity and performance using partitioned global address space languages," in *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*. New York, NY, USA: ACM, 2007, pp. 24–32.

[15] "UPC language specifications, v1.2," Lawrence Berkeley National Lab, Tech. Rep. LBNL-59208, 2005.

[16] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," *ACM Fortran Forum*, vol. 17, no. 2, pp. 1 – 31, 1998. [Online]. Available: citeseer.ist.psu.edu/numrich98coarray.html

[17] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A high-performance java dialect," *Concurrency: Practice and Experience*, vol. 10, no. 11-13, September-November 1998.

[18] Cray Inc., "The Chapel Parallel Programming Language Home Page," `http://chapel.cray.com/` (Mar. 2011).

[19] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt, *The Fortress Language Specification*, 1st ed., Sun Microsystems, Inc., Sep. 2006. [Online]. Available: http://research.sun.com/projects/plrg/fortress.pdf

[20] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 519–538. [Online]. Available: http://doi.acm.org/10.1145/1094811.1094852

[21] R. A. v. d. Geijn and J. Watts, "Summa: scalable universal matrix multiplication algorithm," *Concurrency - Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[22] T. Warburton, "Lecture 24: Brief introduction to block lu factorization and parallel implementation." 2011. [Online]. Available: http://www.caam.rice.edu/~timwar/MA471F03/Lecture24.pdf

[23] D. Campbell, M. Hall, W. Harrod, J. Hiller, D. Koester, J. Levesque, R. Schreiber, and A. Snavely, "Exascale software study : Software challenges in extreme scale systems exascale software study : Software challenges in extreme scale systems," *Government PROcurement*, pp. 1–159, 2009.

A.7 cphVB: A Scalable Virtual Machine for Vectorized Applications

M. Kristensen, S. Lund, T. Blum, and B. Vinter. cphVB: A Scalable Virtual Machine for Vectorized Applications
Submitted to the International Conference on Parallel Processing (ICPP'12). Pittsburgh, PA, 2012.

cphVB: A Scalable Virtual Machine for Vectorized Applications

Mads Ruben Burgdorff Kristensen
Niels Bohr Institute
University of Copenhagen
Denmark
madsbk@nbi.dk

Simon Andreas Frimann Lund
Niels Bohr Institute
University of Copenhagen
Denmark
safl@nbi.dk

Troels Blum
Niels Bohr Institute
University of Copenhagen
Denmark
blum@nbi.dk

Brian Vinter
Niels Bohr Institute
University of Copenhagen
Denmark
vinter@nbi.dk

***Abstract*—Modern processor architectures, in addition to having still more cores, also require still more consideration to memory-layout in order to run at full capacity. The usefulness of most languages is deprecating as their abstractions, structures or objects are hard to map onto modern processor architectures efficiently.**

The work in this paper introduces a new abstract machine framework, cphVB, that enables a vector oriented high-level programming languages to map onto a broad range of architectures efficiently. The idea is to close the gap between high-level languages and hardware optimized low-level implementations. By translating high-level vector operations into an intimidate vector byte code, cphVB enables specialized vector engines to efficiently execute the vector operations.

The primary success parameters are to maintain a complete abstraction from low-level details and to provide efficient code execution across different, modern, processors. We evaluate the presented design through a setup that targets multi-core CPU architectures. We evaluate the performance of the implementation using Python implementations of well-known algorithms: a k-nearest neighbor, a Pi approximation, an n-body simulation, and a shallow water simulation. All demonstrate good performance.

I. Introduction

Obtaining high performance from todays computing environments requires both a deep and broad working knowledge on computer architecture, communication paradigms and programming interfaces. Todays computing environments are highly heterogeneous consisting of a mixture of CPUs, GPUs, FPGAs and DSPs orchestrated in a wealth of architectures and lastly connected in numerous ways.

Utilizing this broad range of architectures manually requires programming specialists and is a very time-consuming task – time and specialization a scientific researcher typically does not have. A high-productivity language that allows rapid prototyping and still enables utilizing of a broad range of architectures is clearly preferable.

There exist high-productivity language and libraries that automatically utilize parallel architectures [1][2][3]. They are however still few in numbers and have one problem in common. They are closely coupled to both the front-end, i.e. programming language and IDE, and the back-end, i.e. computing device, which makes them interesting only to the few using the exact combination of front-, and back-end.

A tight coupling between front-end technology and back-end present another problem; the usefullnes of the developed program expires as soon as the back-end does. With the rapid development of hardware architectures the time spend on implementing optimized programs for a specific hardware target is lost as soon as the hardware product expires.

In this paper, we present a novel approach to the problem of closing the gap between high-productivity languages and parallel architectures, which allows a high degree of modularity and reusability. The approach involves creating a framework, cphVB, in which the computing devices (hardware) are viewed as engines that processes vectorized instructions, called Vector Engines. It defines a clear and easy to understand byte code language that the Vector Engines executes. cphVB also contains a protocol to govern the safe, and efficient exchange, creation, and destruction of model data.

cphVB provides a retargetable framework in which the user can write programs utilizing whichever cphVB supported programming interface they prefer and run the program on their own workstation while doing prototyping, such as testing correctness and functionality of their programs. Users can then deploy the exact same program in a more powerful execution environment without changing a single line of code and thus effectively solve greater problem sets.

The rest of the paper is organized as follows. In Section III we describe the programming model supported by cphVB. Section II describes Numerical Python, which is the first programming interface that fully supports cphVB. In Section IV, we describe the overall design of cphVB. Section V describes an implementation of the cphVB design. In Section VI, we conduct a performance study of the implementation. Finally, in Section VII and VIII we discuss future work and conclude.

A. Related Work

The key motivation for cphVB is to provide a framework for the utilization of heterogeneous computing systems with the goal of obtaining high performance and high productivity. Systems such as pyOpenCL/pyCUDA[4] provides a direct mapping from frontend language to the optimization target. In this case, providing the user with direct access to

the low-level systems OpenCL[5] and CUDA[6] from the high-level language Python[7]. The work in [4] enables the user with the ability to write a low-level implementation combined with a high-productivity language. The goal is similar to cphVB – the approach however is entirely different. cphVB provides a means to hide low-level target specific code behind a programming model and providing a framework and runtime environment to support it.

Intel Math Kernel Library[8] is in this regard more comparable to cphVB. Intel MKL is a programming library providing utilization of multiple targets ranging from a single core CPU to a multi-core shared memory CPU and even to a cluster of computers all through the same programming API. However, cphVB is not only a programming library it is a runtime system providing support for a vector oriented programming model. The programming model is well-known from high-productivity languages such as MATLAB [9], R[10], IDL[11], GNU Octave[12] and Numerical Python (NumPy) [13] to name a few.

cphVB is more closely related to the work described in [14], here a compilation framework is provided for execution in a hybrid environment consisting of both CPUs and GPUs. Their framework uses a Python/NumPy based frontend that uses Python decorators as hints to do selective optimizations. cphVB similarly provides a NumPy based frontend and equivalently does selective optimizations. However, cphVB uses a slightly less obtrusive approach; program selection hints are sent from the frontend via the NumPy-bridge. This approach provides the advantage that any existing NumPy program can run unaltered and take advantage of cphVB without changing a single line of code. Whereas unPython requires the user to manually modify the source code by applying hints in a manner similar to that of OpenMP[15]. This non-obtrusive design at the source level is to the authors knowledge novel.

Microsoft Accelerator[2] introduces ParallelArray, which is similar to the utilization of the NumPy arrays in cphVB but there are strict limitations to the utilization of ParallelArrays. ParallelArrays does not allow the use of direct indexing, which means that the user must copy a ParallelArray into a conventional array before indexing. cphVB instead allows indexed operations and additionally supports array-views, which are array-aliases that provide multiple ways to access the same memory allocation. Thus, the data structure in cphVB is highly flexible and provides elegant programming solutions for a broad range of numerical algorithms. Intel provides a similar approach called Intel Array Building Blocks (ArBB) [3] that provides retargetability and dynamic compilation. It is thereby possible to utilize heterogeneous architectures from within standard C++. The retargetability aspect of Intel ArBB is represented in cphVB as a plain and simple configuration file that define the cphVB runtime environment. Intel ArBB provides a high performance library that utilizes a heterogeneous environment and hides the low-level details behind a vector oriented programming model similar to cphVB. However, ArBB only provides access to the programming model via C++ whereas cphVB is not biased towards any one specific frontend language.

On multiple points cphVB is closely related in functionality and goals to the SEJITS [16] project. SEJITS takes a different approach towards the frontend and programming model. SEJITS provides a rich set of computational kernels in a high-productivity language such as Python or Ruby. These kernels are then specialized towards optimality criteria. This approach has shown to provide performance that at times out-performs even hand-written specialized code towards a given architecture. Being able to construct computational kernels is a core issue in data-parallel programming.

The programming model in cphVB does not provide this kernel methodology. cphVB has a strong NumPy heritage which also shows in the programming model. The advantage is easy adaptability of the cphVB programming model for users of NumPy, Matlab, Octave and R. The cphVB programming model is not a stranger to computational kernels – cphVB deduce computational kernels at runtime by inspecting the vector bytecode generated by the Bridge.

cphVB provides in this sense a virtual machine optimized for execution of vector operations, previous work [17] was based on a complete virtual machine for generic execution whereas cphVB provides an optimized subset.

II. Numerical Python

Before describing the design of cphVB, we will briefly go through Numerical Python (NumPy) [13]. Numerical Python heavily influenced many design decisions in cphVB – it also uses a vector oriented programming model as cphVB.

NumPy is a library for numerical operations in Python, which is implemented in the C programming language. NumPy provides the programmer with an multidimensional array object and a whole range of supported array operations. By using the array operations, NumPy takes advantage of the performance of C while retaining the high abstraction level of Python.

NumPy uses an arrays syntax that is based on the Python list syntax. The arrays are indexed positionally, 0 through length – 1, where negative indexes is used for indexing in the reversed order. Like the list syntax in Python, it is possible to index multiple elements. All indexing that represents more than one element returns a view of the elements rather than a new copy of the elements. It is this view semantic that makes it possible to implement a stencil operation as demonstrated in Figure 1 and 2. In order to force a real array copy rather than a new array reference NumPy provides the `copy` method.

In the rest of this paper, we define the *array-base* as the original allocate array that lies contiguous in memory. In addition, we will define the *array-view* as a view of elements in an array-base.

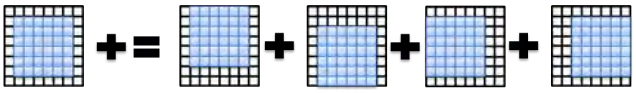


Figure 1. Matrix expression of a simple 5-point stencil computation example. See Figure 2, line 8 for the Python expression.

```
1 center = full[1:-1, 1:-1]
2 up     = full[0:-2, 1:-1]
3 down   = full[2:  , 1:-1]
4 left   = full[1:-1, 0:-2]
5 right  = full[1:-1, 2:  ]
6 while epsilon < delta:
7   work[:] = center
8   work += 0.2 * (up+down+left+right)
9   center[:] = work
```

Figure 2. A 5-Point-Stecil application implemented in Python using NumPy.

III. Target Programming Model

To hide the complexities of obtaining high-performance from a heterogeneous environment any given system must provide a meaningful high-level abstraction. This can be realized in the form of domain specific languages, embedded languages, language extensions, libraries, APIs etc. Such an abstraction serves two purposes: 1) It must provide meaning for the end-user such that the goal of high-productivity can be met with satisfaction. 2) It must provide an abstraction that consists of a sufficient amount of information for the system to optimize its utilization.

cphVB is not biased towards any specific choice of abstraction or frontend technology as long as it is compatible with a vector oriented programming model. This provides means to use cphVB in functional programming languages, provide a frontend with a strict mathematic notation such as APL[18] or a more relaxed syntax such as MATLAB.

The vector oriented programming model encourages expressing programs in the form of high-level array operations, e.g. by expressing the addition of two arrays using one high-level function instead of computing each element individually. The NumPy application in Figure 2 is a good example of using the vector oriented programming model.

IV. Design of cphVB

The key contribution in this paper is a framework, cphVB, that support a vector oriented programming model. The idea of cphVB is to provide the mechanics to seamlessly couple a programming language or library with an architecture-specific implementation of vectorized operations.

cphVB consists of a number of components that communicate using a simple protocol. Components are allowed to be architecture-specific but they are all interchangeable since all uses the same communication protocol. The idea is to make it possible to combine components in a setup that perfectly match a specific execution environment. cphVB consist of the following components:

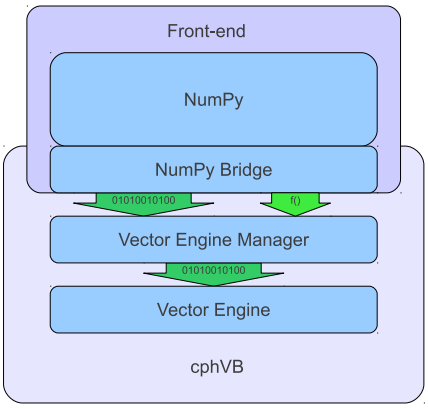


Figure 3. cphVB design idea

- **Programming Interface** The programming language or library exposed to the user. cphVB was initially meant as a computational back-end for the Python library NumPy, but we have generalized cphVB to potential support all kind of languages and libraries. Still, cphVB has design decisions that are influenced by NumPy and its vector objects.
- **Bridge** The role of the Bridge is to introduce cphVB into an already existing languages and libraries. The Bridge generates the cphVB byte code that corresponds to the user-code.
- **Vector Engine** The Vector Engine is the architecture-specific implementation that executes cphVB byte code.
- **Vector Engine Manager** The Vector Engine Manager manages data location and ownership of vectors. It also manages the distribution of computing jobs between potentially several Vector Engines, hence the name.

An overview of the design can be seen in Figure 3.

A. Configuration

To make cphVB as flexible a framework as possible, we manage the setup of all the components at runtime through a configuration file. The idea is that the user can change the setup of components simply by editing the configuration file before executing the user application. Additionally, the user only has to change the configuration file in order to run the application on different systems with different computational resources. The configuration file uses the ini syntax – Figure 4 is an example of a setup for NumPy executing parallel on one machine using Pthreads.

B. Byte Code

The central part of the communication between all the components in cphVB is vector byte code. The goal with the byte code language is to be able to express operations on multidimensional vectors. Taking inspiration from single

```
#Root of the setup
[setup]
bridge = numpy
debug = true

#Bridge for NumPy
[numpy]
type = bridge
children = node

#Vector Engine Manager for a single machine
[node]
type = vem
impl = libcphvb_vem_node.so
children = pthread

#Vector Engine implemented using Posix Threads
[pthread]
type = ve
impl = libcphvb_ve_pthd.so
```

Figure 4. Configuration ini-file

instruction, multiple data (SIMD) instructions but adding structure to the data. This, of course, fits very well with the array operations in NumPy but is not bound to nor limited to these.

We would like the byte code to be a concept that is easy to explain and understand. It should have a simple design that is easy to implement. It should be easy and inexpensive to generate and decode. To fulfill these goals we chose a design that conceptually is an assembly language where the operands are multidimensional vectors. Furthermore, to simplify the design the assembly language should have a one-to-one mapping between instruction mnemonics and opcodes.

In the basic form, the byte-code instructions are primitive operations on data, e.g. addition, subtraction, multiplication, division, square root etc. As an example, let us look at addition. Conceptually it has the form:

```
add $d, $a, $b
```

Where `add` is the opcode for addition. After execution `$d` will contain the sum of `$a` and `$b`.

The requirement is straightforward: we need an opcode. The opcode will explicitly identify the operation to perform. Additionally the opcode will implicitly define the number of operands. Finally, we need some sort of symbolic identifiers for the operands. Keep in mind that the operands will be multidimensional arrays.

C. Interface

The Vector Engine and the Vector Engine Manager exposes simple API that consists of the following functions: initialization, finalization, registration of a user-defined operation and execution of a list of byte codes. Furthermore, the Vector Engine Manager exposes a function to define new arrays.

D. Bridge

The Bridge is the *bridge* between the programming interface, e.g. Python/NumPy, and the Vector Engine Manager. The Bridge is the only component that is specifically implemented for the programming interface. In order to add cphVB support to a new language or library, one only has to implement the bridge component. It generates byte code based on programming interface and sends them to the Vector Engine Manager.

E. Vector Engine Manager

Instead of just letting the front-end communicate directly with the Vector Engine, we introduced a Vector Engine Manager (VEM) into the design. It is the responsibility of the VEM to manage data ownership and distribute byte code instructions to several Vector Engines.

For efficiency reasons, the VEM handles instantiating and discarding arrays. If the programming interface or the Bridge controls this, they would always have to copy data from main memory to the device that is going to do the calculations. Often arrays are created with structured data (e.g. random, constants), with no data at all (e.g. empty), or as a result of calculation. In any case it saves, potentially several, memory copies to delay the actual memory allocation. Typically, array data will exist on the computing device exclusively.

In order to minimize data copying we introduce a data ownership scheme. It keeps track of which components in cphVB that needs to access a given array. The goal is to allow the system to have several copies of the same data while ensuring that they are in synchronization. We base the data ownership scheme on three instructions, *sync*, *release* and *discard*:

- **Sync** is used to request read access to a data object. This means that when acknowledging a *sync* request, the copy existing in shared memory needs to be the most resent copy.
- **Discard** is used to signal that the copy in shared memory has been updated and all other copies are now invalid. Normally used for upgrading a read access to a write access.
- **Release** is simply the same as a *sync* followed by a *discard*. This is used for requesting write access.

The cphVB components follow the following four rules when implementing the data ownership scheme:

1) The Bridge will always ask the Vector Engine Manager for access to a given data object. It will send a *sync* request for read access and a *release* request for write access. The Bridge will not keep track of ownership itself.
2) A Vector Engine can assume that it has write access to all of the output parameters that are referenced in the instructions it receives. Likewise, it can assume read access on all input parameters.

3) A Vector Engine is free to manage its own copies of arrays and implement its own scheme to minimize data copying. It just needs to copy modified data back to share memory when receiving a *sync* instruction and delete all local copies when receiving a *discard* instruction. A *release* instruction can be handled as a*sync* followed by a *discard* instruction.
4) The Vector Engine Manager keeps track of array ownership for all its children. The owner of an array has full (i.e. write) access. When the parent component of the Vector Engine Manager, normally the Bridge, request access to an array, the Vector Engine Manager will forward the request to the relevant child component. The Vector Engine Manager never accesses the array itself.

The Vector Engine Manager also keeps track of how many references there is to any given array. If there are no more references to an array it deallocates memory and sends discard instructions to any child component that may have a local copy.

Additionally, the Vector Engine Manager needs the capability to handle multiple children components. In order to maximize parallelism the Vector Engine Manager can distribute workload and array data between its children components.

F. Vector Engine

Though the Vector Engine is the most complex component of cphVB, it has a very simple and a clearly defined role. It has to execute all instructions it receives in a manner that obey the serialization dependencies between instructions. Finally, it has to ensure that the rest of the system has access to the results as governed by the rules of the *sync*, *release*, and *discard* instructions.

V. Implementation of cphVB

In order to demonstrate our cphVB design we have implemented a basic cphVB setup. This concretization of cphVB is by no means exhaustive. The setup is targeting the NumPy library executing on a single machine with multiple CPU-cores. In this section, we will describe the implementation of each component in the cphVB setup – the Bridge, the Vector Engine Manager, and the Vector Engine. The cphVB design rules (Sec. IV) govern the interplay between the components.

A. Bridge

The role of the Bridge is to introduce cphVB into an already existing project. In this specific case NumPy, but could just as well be "R" or any other language/tool that works primarily on vectorizable operations on large data objects.

It is the responsibility of the Bridge to generate cphVB instructions on basis of the Python program that is being run. The NumPy Bridge is an extension of NumPy version 1.6. It uses hooks to divert function call where the program access cphVB enabled NumPy arrays. The hooks will translate a given function into its corresponding cphVB byte code when possible. When it is not possible, the hooks will feed the function call back into NumPy and thereby forcing NumPy to handle the function call itself.

The Bridge operates with two address spaces for arrays: the cphVB space and the NumPy space. All arrays starts in the NumPy space as a default. The original NumPy implementation handles these arrays and all operations using them. It is possible to assign an array to the cphVB space explicitly by using an optional cphVB parameter in array creation functions such as `empty` and `random`. The cphVB bridge implementation handles these arrays and all operations using them.

In two circumstances, it is possible for an array to transfer from one address space to the other implicitly at runtime.

1) When an operation accesses an array in the cphVB address space but it is not possible for the bridge to translate the operation into cphVB code. In this case, the bridge will synchronize and move the data to the NumPy address space. For efficiency no data is actually copied instead the bridge uses the `mremap`[1] function to re-map the relevant memory pages.
2) When an operations access arrays in different address spaces the Bridge will transfer the arrays in the NumPy space to the cphVB space. Afterwards, the bridge will translate the operation into byte code that cphVB can execute.

In order to detect direct access to arrays in the cphVB address space by the user, the original NumPy implementation, a Python library or any other external source, the bridge protects the memory of arrays that are in the cphVB address space using `mprotect`[2]. Because of this memory protection, subsequently accesses to the memory will trigger a segmentation fault. The Bridge can then handle this kernel signal by transferring the array to the NumPy address space and cancel the segmentation fault. This technique makes it possible for the Bridge to support all valid Python/NumPy application since it can always fallback to the original NumPy implementation.

In order to gather greatest possible information at runtime, the Bridge will collect a batch of instructions rather than executing one instruction at a time. The Bridge will keep recording instruction until either the application reaches the end of the program or untranslatable NumPy operations forces the Bridge to move an array to the NumPy address space. When this happens, the Bridge will call the Vector Engine Manager to execute all instructions recorded in the batch.

[1]The function `mremap()` in GNU C library 2.4 and greater.
[2]The function `mprotect()` in the POSIX.1-2001 standard.

Processor	Two Intel Xenon
Clock	2.67 GHz GHz
Private L1 Data Cache	64 KB
Private L2 Data Cache	512 KB
Shared L3 Cache per Socket	12MB
Memory Bandwidth	25.6 GB/s
Memory per Node	96GB DDR3-1066
Compiler	GCC 4.4.5

Table I
LENOVO THINKSTATION D20

B. Vector Engine Manager

The Vector Engine Manager (VEM) in our setup is very simple because it only has to handle one Vector Engine thus all operations go to the same Vector Engine. Still, the VEM creates and deletes arrays based on specification from the Bridge and handles all meta-data associated with arrays.

C. Vector Engine

In order to maximize the CPU cache utilization and enables parallel execution the first stage in the VE is to form a set of instructions that enables data blocking. That is, a set of instructions where all instructions can be applied on one data block completely at a time without violating data dependencies. This set of instructions will be referred to as a kernel.

The VE will form the kernel based on the batch of instructions it receives from the VEM. The VE examines each instruction sequentially and keep adding instruction to the kernel until it reaches an instruction that is not *blockable* with the rest of the kernel. In order to be blockable with the rest of the kernel an instruction must satisfy the following two properties where A is all instructions in the kernel and N is the new instruction.

1) The input arrays of N and the output array of A do not share any data or represents precisely the same data.
2) The output array of N and the input and output arrays of A do not share any data or represents precisely the same data.

When the VE has formed a kernel, it is ready for execution. Since all instruction in a kernel supports data blocking the VE can simply assign one block of data to each CPU-core in the system and thus utilizing multiple CPU-cores. In order to maximize the CPU cache utilization the VE may divide the instructions into even more data blocks. The idea is to access data in chunks that fits in the CPU cache. The user, through an environment variable, manually configures the number of data blocks the VE will use.

VI. PERFORMANCE STUDY

In order to demonstrate the performance of our initial cphVB implementation and thereby the potential of the cphVB design, we will conduct some performance benchmarks using NumPy[3]. We execute the benchmark applications on one Lenove ThinkStation D20 with two Intel Xenon processers (Table I).

We execute the benchmark applications on one Lenovo ThinkStation D20 with two Intel Xenon processers (Table I). The experiments use up to all eight CPU-cores on the machine and for each executaion we calculate the speedup of cphVB compared to NumPy. We perform strong scaling experiments, in which the problem size is constant though all the executions. For each experiment, we find the block size that results in best performance and we calculate the result of each experiment using the average of three executions.

The benchmark consists of the following Python/NumPy applications. All are pure Python applications that make use of NumPy and none uses any external libraries.

- **Monte Carlo Pi** Approximating Pi using Monte Carlo simulation. The implementation is a translation and vectorization of the Monte Carlo simulation included in the benchmark suite SciMark 2.0[19], which is written in Java (Fig. 5).
- **kNN** A naïve implementation of a k Nearest Neighbor search (Fig. 6).
- **N-body** A Newtonian N-body simulation that uses a $O(n^2)$ algorithm that computes all body-body interactions. (Fig. 7).
- **Shallow Water** A simulation that simulates a system governed by the shallow water equations. It is a translation of a MATLAB application by Burkardt[20] (Fig. 8).

A. Discussion

The Monte Carlo Pi simulation is an embarrassingly parallel problem because thread coordination is only relevant at the end of the program. Thus, cphVB provides good performance speedup compared to NumPy – at eight CPU-cores cphVB is more than six times faster than NumPy.

On the other hand, our naïve implementation of the k Nearest Neighbor search is not an embarrassedly parallel problem. However, it has a time complexity of $O(n^2)$ when the number of elements and the size of the query set is n, thus the problem should be scalable. The result of our experiment is also promising – with a performance speedup of more than five when running on eight CPU-cores. Because of better cache utilization through data blocking, the performance of cphVB is more than twice as good as NumPy when using one CPU-core. Still, when using more than four CPU-cores the memory bandwidth becomes the limiting factor.

The N-body simulation also has a time complexity of $O(n^2)$ but it exhibits less cache locality. At one CPU-core NumPy outperforms cphVB by quite a margin. This is

[3]NumPy version 1.6.1

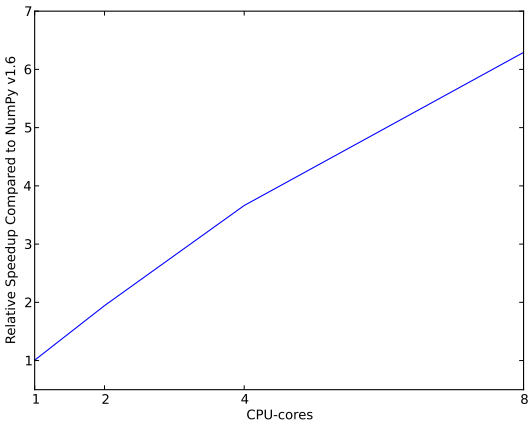


Figure 5. Runtime of the Monte Carlo Pi Approximation. The job consists of a vector with 100M elements using 10 iterations.

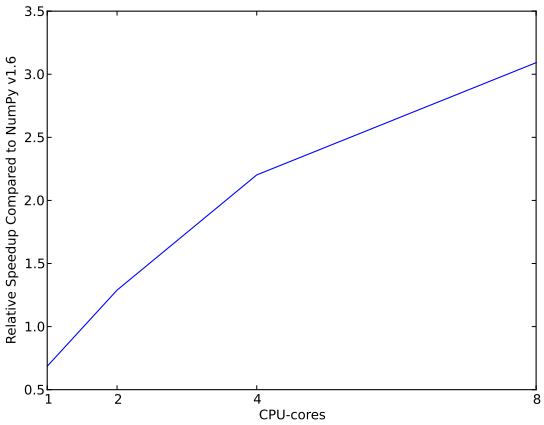


Figure 7. Runtime of the N-body simulation. The job consists of 8K bodies that simulate 10 time steps.

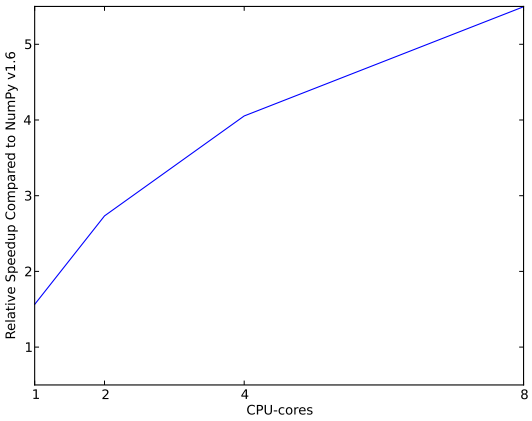


Figure 6. Runtime of the k Nearest Neighbor search. The job consists of 1K elements and the query set also consists of 1K elements.

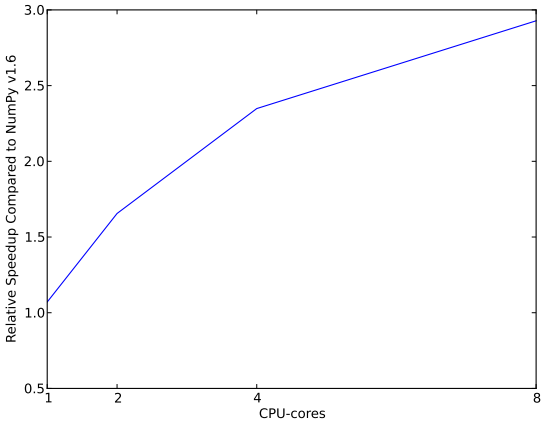


Figure 8. Runtime of the Shallow Water Equation. The job consists of 4M grid points that simulate 10 time steps.

because the N-body implementation uses matrix transpose as part of the computation loop, which NumPy controls more efficiently than cphVB.

Finally, the Shallow Water simulation only has a time complexity of $O(n)$ thus it is the most memory intensive application in our benchmark. Still, cphVB manages to achieve a performance speedup of almost three compared to NumPy.

VII. Future Work

The future goals of cphVB involves improvement in two major areas; expanding support and improving performance. Work has started on a CIL-bridge which will leverage the use of cphVB to every CIL based programming language which among others include: C#, Visual C++ and VB.NET. Another project in current progress within the area of support is a C++ bridge providing a library-like interface to cphVB using overloading and templates to provide a high-level interface in C++.

To improve both support and performance, work is in progress on a vector engine targeting OpenCL compatible hardware, mainly focusing on using GPU-resources to improve performance. Additionally the support for program execution using distributed memory is on progress. This functionality will be added to cphVB in the form a vector engine manager.

In terms of pure performance enhancement, cphVB will introduce JIT compilation in order to improve memory intensive applications. The current vector engine for multi-cores CPUs uses data blocking to improve cache utilization but as our experiments show then the memory intensive applications still suffer from the von Neumann bottleneck[21].

By JIT compile the instruction kernels, it is possible to improve cache utilization drastically.

VIII. Conclusion

The vector oriented programming model used in cphVB provides a framework for high-performance and high-productivity. It enables the end-user to execute vectorized applications on a broad range of hardware architectures efficiently without any hardware specific knowledge. Furthermore, the cphVB design supports scalable architectures such as clusters and supercomputers. It is even possible to combine architectures in order to exploit hybrid programming where multiple levels of parallelism exist. The authors in [22] demonstrate that combining shared memory and distributed memory parallelism through hybrid programming is essential in order to utilize the Blue Gene/P architecture fully.

In a case study, we demonstrate the design of cphVB by implementing a frontend for Python/NumPy that targets multi-core CPUs in a shared memory environment. The implementation executes vectorized applications in parallel without any user intervention. Thus showing that it is possible to retain the high abstraction level of Python/NumPy while fully utilizing the underlying hardware. Furthermore, the implementation demonstrates scalable performance – a k-nearest neighbor search purely written in Python/NumPy obtains a speedup of more than five compared to a native execution.

Future work will further test the cphVB design model as new frontend technologies and heterogeneous architectures are supported.

References

[1] M. R. B. Kristensen and B. Vinter, "Numerical Python for Scalable Architectures," in *Fourth Conference on Partitioned Global Address Space Programming Model, PGAS{'}10*. ACM, 2010. [Online]. Available: http://distnumpy.googlecode.com/files/kristensen10.pdf

[2] T. David, P. Sidd, and O. Jose, "Accelerator : Using Data Parallelism to Program GPUs for General-Purpose Uses," *October*. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=70250

[3] C. J. Newburn, B. So, Z. Liu, M. Mccool, A. Ghuloum, S. D. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang, "Intel s Array Building Blocks : A Retargetable , Dynamic Compiler and Embedded Language," *Symposium A Quarterly Journal In Modern Foreign Literatures*, pp. 1–12, 2011. [Online]. Available: http://software.intel.com/en-us/blogs/wordpress/wp-content/uploads/2011/03/ArBB-CGO2011-distr.pdf

[4] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation," *Brain*, vol. 911, no. 4, pp. 1–24, 2009. [Online]. Available: http://arxiv.org/abs/0911.3456

[5] K. Opencl, W. Group, and A. Munshi, "OpenCL Specification," *ReVision*, pp. 1–377, 2010. [Online]. Available: http://scholar.google.com/scholar?hl=en\&btnG=Search\&q=intitle:OpenCL+Specification\#2

[6] N. Nvidia, "NVIDIA CUDA Programming Guide 2.0," pp. 1–111, 2010. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf

[7] G. V. Rossum and F. L. Drake, "Python Tutorial," *History*, vol. 42, no. 4, pp. 1–122, 2010. [Online]. Available: http://docs.python.org/tutorial/

[8] Intel, "Intel Math Kernel Library (MKL)," pp. 2–4, 2008. [Online]. Available: http://software.intel.com/en-us/articles/intel-mkl/

[9] MATLAB, *version 7.10.0 (R2010a)*. Natick, Massachusetts: The MathWorks Inc., 2010.

[10] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2011. [Online]. Available: http://www.r-project.org

[11] B. A. Stern, "Interactive Data Language." ASCE, 2000.

[12] J. W. Eaton, "GNU Octave," *History*, vol. 103, no. February, pp. 1–356, 1997. [Online]. Available: http://www.octave.org

[13] T. E. Oliphant, "Python for Scientific Computing," *Computing in Science Engineering*, vol. 9, no. 3, pp. 10–20, 2007. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4160250

[14] R. Garg and J. N. Amaral, "Compiling Python to a hybrid execution environment," *Computing*, pp. 19–30, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?id=1735688.1735695

[15] R. V. D. Pas, "An Introduction Into OpenMP," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 5, pp. 1–82, 2005. [Online]. Available: http://portal.acm.org/citation.cfm?id=1168898

[16] B. Catanzaro, S. Kamil, Y. Lee, K. Asanović, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and O. Fox, "SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization," in *Proc of 1st Workshop Programmable Models for Emerging Architecture PMEA*, no. UCB/EECS-2010-23, EECS Department, University of California, Berkeley. Citeseer, 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-23.html

[17] R. Andersen and B. Vinter, "The Scientific Byte Code Virtual Machine," in *Proceedings of the 2008 International Conference on Grid Computing & Applications, GCA 2008 : Las Vegas, Nevada, USA, July 14-17, 2008. CSREA Press.*, 2008, pp. 175–181. [Online]. Available: http://dk.migrid.org/public/doc/published_papers/sbc.pdf

[18] "why apl?" [Online]. Available: http://www.sigapl.org/whyapl.htm

[19] R. Pozo and B. Miller, “SciMark 2.0,” 2002. [Online]. Available: http://math.nist.gov/scimark2/

[20] J. Burkardt, “Shallow Water Equations,” 2010. [Online]. Available: http://people.sc.fsu.edu/~jburkardt/m_src/shallow_water_2d/

[21] J. Backus, “Can Programming be Liberated from the von Neumann Style?: A Functional Style and its Algebra of Programs,” *Communications of the ACM*, vol. 16, no. 8, pp. 613–641, 1978.

[22] M. Kristensen, H. Happe, and B. Vinter, “Hybrid Parallel Programming for Blue Gene/P,” *Scalable Computing: Practice and Experience*, vol. 12, no. 2, pp. 265–274, 2011.